University of Twente
The Netherlands

Automated transformations from ECA rules to Jess

NAME	:	N.C. Maatjes
STUDENT NUMBER	:	s0040495
PERIOD	:	4-12-2006 until 13-7-2007
DATE	:	13-7-2007
SUPERVISOR	:	L. Ferreira Pires
GRADUATION COMMITTEE	:	L. Ferreira Pires
		P. Dockhorn Costa
		M.J. van Sinderen

Abstract

Context-awareness is the phenomenon in computer science in which portable devices are able to keep track of their context, of the user's surroundings, in order to supply the user with relevant services. Applications using context and delivering such services are called "context-aware applications".

The Event-Control-Action (ECA) architectural pattern can be used to give structure to such context-aware applications by specifying a way for them to process their data and act upon it by means of simple if-then constructs.

In order to specify such if-then constructs, which are also called rules, the ECA Domain-specific Language (ECA-DL) has been developed. Currently, there exists no interpreter for this language.

The Java Expert System Shell (Jess) is a rule engine for Java. A rule engine is a program that tries to match rules against information and then triggers one or more actions. This rule engine supports the Jess language, which is a declarative language. This means commands in the Jess language state only *what* the computer should do, and not exactly *how* it should do this.

This thesis presents how the Model-Driven Architecture (MDA) approach can be used to develop a mapping between ECA-DL and the Jess language, and how to set up an automated transformation from ECA-DL rules to Jess rules, using this mapping. This automated transformation facilitates the implementation of context-aware applications that implement the ECA pattern, by providing a means for using ECA-DL rules in an already existing interpreter, i.e., the Jess engine.

Preface

This master's thesis is the result of the research done for the master's assignment at the Architecture and Services of Network Applications (ASNA) group at the University of Twente. The research has been carried out from December 2006 until July 2007.

I would like to express my gratitude to my supervisors, L. Ferreira Pires, M. van Sinderen and P. Dockhorn Costa, for their help, suggestions and comments on my work, as well as their appreciation of my work.

I would also like to thank my family, friends and colleagues for the time they spent reading and reviewing my work.

Lastly, I would like to thank I. Kurtev and A. Goknil, whose help during the project is highly appreciated.

N.C. Maatjes

Acknowledgement

This work is part of the Freeband A-MUSE project (http://a-muse.freeband.nl), which is sponsored by the Dutch government under contract BSIK 03025.

Table of contents

1. Introduction

In this chapter, we present the motivation, the goals, the approach and the structure of this thesis. We introduce context-awareness and some problems that are addressed in this project. We also present some research initiatives related to this project.

1.1. Motivation

In recent years, computing devices have become small and powerful enough to be taken with at all times. This opens up new possibilities, because the context of the device has moved from static (i.e., computers remained in the office) to dynamic (a laptop or PDA, or even smaller devices, can easily be taken with).

Context-awareness is the phenomenon in computer science in which such portable devices are able to keep track of their context, of the user's surroundings, in order to supply the user with relevant services.

A context-aware application can keep track of relevant information about the situation of the user, the application itself, or other entities relevant to the interaction between the user and the application, and then act upon this information.

In an office setting, e.g., when an employee enters his office, his PDA could signal the lights to turn on. This situation is displayed in Figure 1. When the employee is traveling, it could give the local weather forecast when he enters another region.

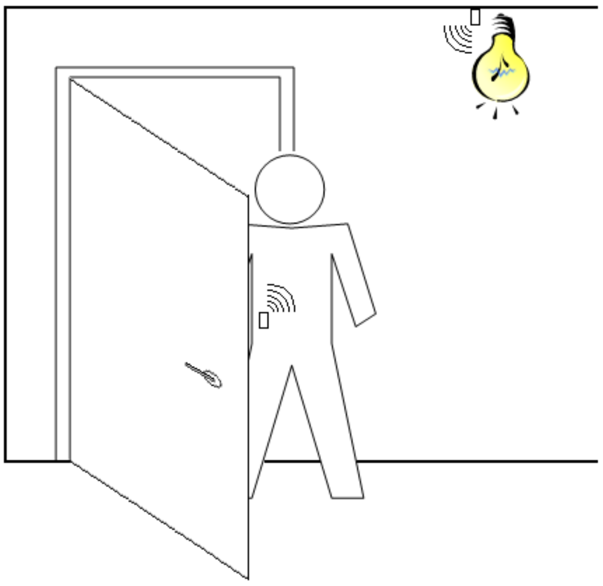


Figure 1: The light turns on when the user enters his office

Context-awareness can also be used in more critical situations. In a hospital e.g., a patient can have a small device with sensors on his body that keeps track of his vital signs, adjusts medication to the patient's needs or even alerts a doctor when something is going wrong.

In order for a context-aware application to respond properly to the various inputs and to be able to cope with new or different rules and information, and in order to avoid having to invent the wheel over and over again, we need a general architectural pattern that can be used in several context-aware applications.

One such pattern is the Event-Control-Action (ECA) pattern [1, 2]. This pattern specifies a way for a context-aware application to process its data and act upon it by means of simple if-then constructs. These if-then constructs are called rules. A rule in the ECA pattern consists of three parts. The first part, the Event, models a change of situation, e.g., someone entering a room. The second part, the Control, makes sure additional conditions are satisfied, e.g., the room is dark. The third part, the Action, executes a command, e.g., the light is turned on.

Following this pattern, a specification language has been defined to specify ECA rules, called ECA Domain-specific Language (ECA-DL) [1, 3], which is thus especially suitable for the domain of context-awareness. However, there is currently no interpreter for ECA-DL rules, which means the rules cannot be directly executed by an application. This problem can be solved by transforming ECA-DL rules to another language, for which there is an interpreter available.

1.2. Goals

The overall goal of this project is to facilitate the implementation of context-aware applications that implement the ECA pattern, by providing a means for using ECA-DL rules in an already existing interpreter. In this sense, we refer to ECA-DL rules as abstract and to rules that can be executed directly as concrete.

The transformation of ECA-DL rules to concrete rules can be implemented by developing a mapping between ECA-DL rules and concrete rules. Such a mapping can then be used in a transformation tool, which performs the actual transformation.

In [4] a start has already been made towards this goal. It uses the rule engine Jess, which is "a fast and powerful rule engine for the Java platform". It also provides a few examples of manual mappings from ECA-DL models to Jess rules.

The main goal of this master's project is to formalize the transformation process from ECA-DL rules to Jess rules and implement this transformation in such a way that it can be automatically executed by a transformation engine.

For the development of this transformation process, we make use of the Model Driven Architecture (MDA) [5] approach. This means rules are expressed using models, which allows us to use a variety of modeling and transformation tools. MDA also allows for a higher level of abstraction, making it easier to create, e.g., another transformation from or to ECA-DL or Jess.

The main goal of this project consists of three subgoals:

- Investigate how ECA-DL can be mapped to Jess and evaluate the alternatives;
- Implement and test some of these alternatives;
- Evaluate the use of automated transformation of abstract rules to concrete rules.

1.3. Approach

In order to realize these goals, we have taken the following approach:

- We have studied ECA-DL and the Jess language and described their workings and syntax;
- We have studied the MDA approach and have described how this approach works and why it is useful for this project;
- We have set up a modeling environment, which allowed us to create models and transformation specifications;
- We have constructed the ECA-DL and Jess metamodels;
- We have investigated what mappings from the ECA-DL metamodel to the Jess metamodel are possible, and have evaluated their usefulness;
- Following the MDA approach, we have implemented and tested some of these alternatives using a transformation tool and corresponding transformation language;
- We have evaluated the use of automated transformation for rules. Because the transformation is only semi-automated, some manual activities are also required. We have investigated why and to what extent manual activities are necessary.

1.4. Related work

In this project, we continue the work started in [4]. Below we mention a few other research initiatives, also addressing this topic.

A good example of a thesis about context-awareness is [6]. This thesis introduces some of the problems with context-aware applications, and proposes "a software architecture to support the building of context-aware applications". This framework makes it easier for context-aware applications to process their data. This framework relates to the ECA pattern, since it also proposes to give structure to context-aware applications. Our project, on the contrary, mainly focuses on the transformation of rules used in a context-aware application, to another language.

The work presented in [7] introduces a "framework for the knowledge-based support of context-awareness in complex systems". This framework uses a knowledge base to integrate information models in order to obtain higher-level information. This integration is performed using rule-based transformation of these information models. As opposed to our project, it focuses on extracting information from several models, instead of transforming these models to another language. Also, the transformation process makes use of multiset transformation, whereas our project uses model transformation as specified by the MDA approach.

The model transformation approach presented in [8] focuses on model transformation in the scope of MDA, and in particular on making transformations more adaptable. Usually, several different transformations between models are possible, yielding target models that are functionally the same, but may differ in adaptability, performance in time and space, etc. [8] proposes that software engineers should be able to choose exactly that transformation that yields the result most suitable for their particular purpose. It thus mainly focuses on the transformation process itself, and on how it can be *improved*. Our project, however, focuses on *using* model transformation as a way to transform an ECA-DL rule to a Jess rule.

The work presented in [9] focuses on different model transformation techniques and shows examples of a transformation between models using the Tefkat transformation language in an Eclipse plugin. Similar to our project, this thesis uses Ecore, the Eclipse variant of MOF. In our project, we have decided to use another transformation language, namely ATL, which offers better documentation and support. Also, our project focuses on ECA-DL and Jess, neither of which are addressed in [9].

The work presented in [10] closely relates to our work. It focuses on a model-driven design trajectory for context-aware and mobile services. A part of this trajectory is the transformation of models from the service specification using ECA-DL, to the platform-independent service design using the A-MUSE abstract platform. This platform-independent service design is then transformed to form the platform-specific realization using Web Services or CORBA. This part of the trajectory is quite similar to our project. Both approaches use ECA-DL as their source language, and use model-driven development to transform source models to a target language. The main difference with our project lies in the fact that this thesis *describes* the trajectory to be used, while we *use* a similar trajectory and develop an actual transformation specification.

1.5. Structure

The structure of this thesis is as follows:

- In chapter 2 we present the technological background of this project, ECA-DL and the Jess language used, as well as the MDA approach. Also, we present the general transformation process and the environment that is being used.
- In chapter 3 we present the ECA-DL and Jess metamodels and we show how they are constructed;
- In chapter 4 we present several mappings between the ECA-DL and Jess metamodels and we evaluate the alternatives;
- In chapter 5 we present the transformation specification and implementation of some of the alternatives, using a transformation tool and language;
- In chapter 6 we present a use case. This use case shows how we can use the developed transformation specification to transform an ECA-DL rule to a Jess rule;
- In chapter 7 we discuss the use of automated transformation rules, present the conclusions of this thesis and identify some topics for future work.

2. Technological background and approach

In this chapter we present the workings and the syntax of ECA-DL and the Jess language, in order to make the reader more acquainted with these languages. The focus is on the general workings of the languages and on those parts that are used in the transformation process. We also present the MDA approach, as a prelude to the transformation process itself, as well as the development environment used in this project.

2.1. ECA-DL

In order to address general problems in the field of computer applications, we can make use of architectural patterns, which provide a general structure for an application and defines how it should behave. In the field of context-awareness, one such architectural pattern is the Event-Control-Action (ECA) pattern, which specifies how a context-aware application should react to changes in its context. Take the following example:

When Pete enters his office, his computer should turn on if it wasn't turned on already.

The ECA pattern divides this process into three parts, the Event, the Control and the Action.

- The Event part monitors the application's context and generates events when the context changes. In this case, the context change is denoted by Pete entering his office, so the event "Pete enters his office" is generated.

- The Control part receives this event and looks for rules that match the event. It iterates over all these rules, checking if any additional conditions are satisfied. If so, the Action part is given a signal to trigger the action specified in the rule. In this case, the Control receives the event and it finds the rule "When Pete enters his office, his computer should turn on if it wasn't turned on already." If Pete's computer is indeed not turned on already, the signal "turn on Pete's computer" is sent to the Action part.

- The Action part, when it receives such a signal, triggers the actions. In this case, it simply turns Pete's computer on.

In the scope of the AWARENESS project [11] the ECA Domain-specific Language (ECA-DL) has been defined, which can be used to specify ECA rules.

ECA-DL Syntax

ECA-DL rules consist of three parts, namely Event, Condition and Action. Each of these parts are represented in the syntax on a separate line by so-called tokens (a "primitive block of structured text", usually a single word) and their arguments. Syntax is denoted using the `Courier New` font, non-literal tokens are placed between angle brackets. The three parts are:

- Event: `Upon <event>`

- Condition: `When <condition>`

- Action: `Do <action>`

Each token here has a different argument:

- `<event>`: This argument contains an event, which consists of one or more state transitions. There are three possible states: true, false and unknown. A transition is denoted using functions like `EnterTrue(inOffice)` or `FalseToUnknown(lightOn)`, i.e., it indicates change. Multiple transitions can be tied by logical operators like AND, OR and NOT to create a compound transition. Only when the entire compound expression has evaluated to true, i.e., when the required transitions have happened, is the event generated and can a number of actions get triggered.
  While transitions can contain expressions, they cannot contain other transitions. Also, it is important to stress that a transition merely indicates the change of a state, while an event actually triggers an action.

- `<condition>`: This argument contains a condition that has to be fulfilled in order for the actions to be triggered. Whereas transitions occur at a certain point in time, conditions hold for certain periods. Examples of conditions are `Pete.height < Mary.height` and `isTurnedOff (computer) AND belongsTo(computer, Pete)`. Conditions (which are expressions) can contain other expressions.

- `<action>`: This argument contains one or more actions to be triggered when an event takes place and the condition is satisfied. In context-awareness, an action is often a notification, which, e.g., can be sent to a student's supervisor to notify him/her that the student has arrived at university. An example action would be `Notify(Supervisor, "Student x has entered the building!")`.

A simple ECA-DL rule, based on the example given above, would thus be:

```
Upon EnterTrue(isInRoom(Pete, office))
When isTurnedOff(computer) AND belongsTo(computer, Pete)
Do turnOn(computer)
```

This example ECA-DL rule states, that when Pete enters his office, and the computer that belongs to Pete is turned off, that his computer should be turned on.

Besides this basic syntax, ECA-DL supports a few other constructs. These are:

- `<Lifetime>`: An additional statement can be used to indicate the lifetime of the rule. The following lifetimes are supported:

  - `always` => activate the rule and keep it active. This is the default;

  - `once` => activate the rule and deactivate the rule after it has been used;

  - `<n> times` => activate the rule deactivate it after it has been used <n> times;

    - `from <start> to <end>` => activate the rule at <start> and deactivate it at <end>. <start> and <end> are moments in time, e.g. “May 1st, 2007”;

    - `to <end>` => activate the rule, and deactivate the rule at <end>;

    - `frequency <n> times per <period>` => activate the rule, and keep it active as long as it has been used fewer than <n> times during <period>. Deactivate and keep the rule inactive as long as it has been used <n> times during <period>.

- `Scope(<collection>, <variable>) { <Basic ECA-DL rule> }`: This clause is used for parameterization, i.e., for defining multiple rules at once. `<variable>` can then be used in the `<Basic ECA-DL rule>` as a reference to each entry in `<collection>`. `Scope` constructs cannot be nested.

- `Select(<collection>, <variable>, <condition>)`: This clause is used for selecting a set of entities matching a condition. For example, select all the people in a certain room using `Select(people.*, person, inRoom(person, office))`. This clause is often used in combination with the `Scope` clause above.

An ECA-DL rule containing these extra constructs could be:

```
Scope(Select(people.*, person, person.age > 30), p)
{
  Upon EnterTrue(isInRoom(p, office))
  isTurnedOff(computer) AND belongsTo(computer, p)
  Do turnOn(computer)
  once
}
```

This creates rules for all people older than 30 years. When such a person enters his office, his computer turns on, but only once. After this, the rule is deactivated only for this person.

For a more detailed description of the ECA pattern and ECA-DL, we refer to [4].

2.2. Jess

As our target system we use the Java Expert System Shell [12] (Jess), a rule engine for Java, which supports the Jess language. A rule engine (also called inference engine) is a program that tries to match rules against information and then triggers one or more actions.

Unlike programs written in an imperative language like C or Java, Jess programs do not specify *how* the computer should behave (e.g., read input, print output, etc.), they only specify *what* the computer should do. It is then the responsibility of the Jess engine to find out *how* to do best *what* is declared. Jess is therefore known as a declarative language.

A Jess program itself can contain two kinds of declarations, namely facts and rules.

- A fact contains basic information, e.g. “Pete is male” or “Mary is the mother of Tom”. This information is stored in working memory;

- A rule contains a simple if-then construct containing statements about facts, e.g. "If Pete is male AND Pete has a child, then Pete is father". This information is stored in the knowledge base.
  The if-part of a rule is called Left-Hand Side (LHS), the then-part is called Right-Hand Side (RHS).

When the Jess engine runs, it looks for matches between the rules and the facts. When such a match has been found, the rule is fired and one or more actions are triggered. This process is repeated as long as changes occur in the working memory or knowledge base, because:

- At a given time, more than one rule may match. Other rules also have to be fired;

- The RHS of a rule can declare new facts or rules, or destroy old ones. New rules also have to be evaluated.

The default strategy for deciding which rules to fire next is the depth strategy, which works on a Last In First Out (LIFO) basis. This means newer rules are activated before older rules. The user can define other strategies, either globally or on a per-rule basis.

Besides supporting facts and rules, the Jess language has much more functionality, like arithmetic or interactions with Java. For a more detailed description of the Jess language, we refer to [4] and [12].

Jess Syntax

When it comes to syntax, Jess is very similar to Lisp. All code in Jess is written using function calls, which are always surrounded by parentheses. After the command, one or more arguments can be given.

In Jess, there are three kind of facts: ordered facts, unordered facts and shadow facts. In this project, we use only ordered and unordered facts.

An unordered fact is created using two function calls. First, we need to create a template for a fact, much like we need to create a Java class before we can instantiate it. A template has a name and a number of slots for information. A basic template looks as follows:

```
(deftemplate Ball (slot size) (slot color))
```

This defines a template called "Ball" with two properties, namely size and color. The following command declares (i.e., instantiates and adds) a fact:

```
(assert (Ball (size 3) (color blue)))
```

Ordered facts are created using only one function call. We do not need to define a template for ordered facts, because Jess does this implicitly. Ordered facts are useful when we do not want to bother with attribute names, or want the attributes to have a fixed order. Declaring an ordered fact looks as follows:

```
(assert (List 3 5 7))
```

This command declares an ordered fact called "List", whose attributes have a fixed order.

Like unordered facts, rules also need to be defined before they can be used. In Jess, a rule consists of a name, an if-part (LHS) and a then-part (RHS). The name must contain a single identifier, which can be used for destroying the rule. The LHS must contain zero or more conditions, which are matched against changed or new facts in the working memory. This means that facts that do not change, also do not trigger rules. The LHS can also contain a few options, like giving higher or lower priority to a rule. The RHS must contain zero or more function calls. The following rule looks for new or changed Balls and prints their properties. The variables `?size` and `?color` contain the actual properties:

```
(defrule printBalls
  (Ball (size ?size) (color ?color))
  =>
  (printout t "Found Ball (" ?size ", " ?color ")" crlf)
)
```

As mentioned before, it is also possible to have a rule declare new facts:

```
(defrule makeMoreBalls
  (Ball (size ?size) (color ?color))
  =>
  (assert (Ball (size (+ ?size 1)) (color ?color)))
)
```

The problem with this rule is that it creates a new Ball, which immediately fires the same rule, ad infinitum.

2.3. MDA

During the last decades, new technologies and programming languages have been designed and old ones have faded away. This has resulted in a lot of so-called legacy code, written in languages or on systems that are no longer supported. Converting this legacy code to a newer technology platform is often very costly, if not impossible. This poses serious problems when legacy code has to work together with newer applications. Even if it is possible to make such a conversion, a few years later an even better technology platform may come along, requiring yet another conversion.

To cope with these problems, the Object Management Group (OMG) has created the Model Driven Architecture [5] (MDA). The main idea behind MDA is that applications, middleware, etc. are no longer directly written in a programming language like Java or C#, but are instead specified as Platform-Independent Models (PIM). This introduces a separation of concerns between the design of an application, and the architecture on which the application will run. Such a design model can then be transformed to a Platform-Specific Model (PSM), which makes use of specific features of a platform. This PSM can then be transformed to code, to an actual implementation.

When a newer, better technology platform becomes available, we need only regenerate the PSM and actual implementation, avoiding a costly manual conversion.

Model transformation

From the section above, it should be clear that model transformation plays an important role in MDA. It is this kind of model transformation that we can use in this project.

Often, rules or languages are translated textually. This means that the input rule is parsed, the tokens are read and transformed, and the output rule is written. Such an implementation is hard to extend or reuse when yet another transformation needs to be made. Also, we have to deal with the syntax and structure at the same time.

Instead, we take a different approach. Rules are no longer treated as text, but they are converted to models. Such a model may, e.g., contain the entities Pete and Computer, linked by the association belongsTo. In this project we use XML Metadata Interchange (XMI) [13] as our modeling language, an integration of the data standards XML, UML and MOF.

So instead of transforming the rules themselves, the models are transformed instead. Although this is similar to textual transformation, there is the advantage of separation of concerns. First we transform the structure of the language, and only later do we deal with the specific syntax.

In this project, we follow the Meta-Object Facility (MOF) [14] standard as specified by the OMG, which specifies the modeling architecture as shown in Figure 2. In this architecture, the first step towards the model-based transformation process is defining what models are allowed to contain, and how entities in a model are related.

In this project, we use the Unified Modeling Language (UML) [15] to create the so-called metamodels. Such a metamodel acts as a mold for models. Just as a rule is written using a language as its mold, a model is written using a metamodel as its mold. Another good analogy here is a natural language. A sentence like “Pete owns this computer” is written using the language English, which compares to a model being written using a metamodel.

Although the English language from this example is self-describing (it defines what is allowed and what is not by using itself), metamodels are not. We thus need one more level for a metametamodel. This metametamodel not only specifies what metamodels are allowed to contain, but is also self-describing (self-conforming). Another level is thus not needed. The OMG has specified the Meta-Object Facility (MOF) as their metametamodel.

This structure of models, metamodels and metametamodels is clarified in the diagram shown in Figure 2.

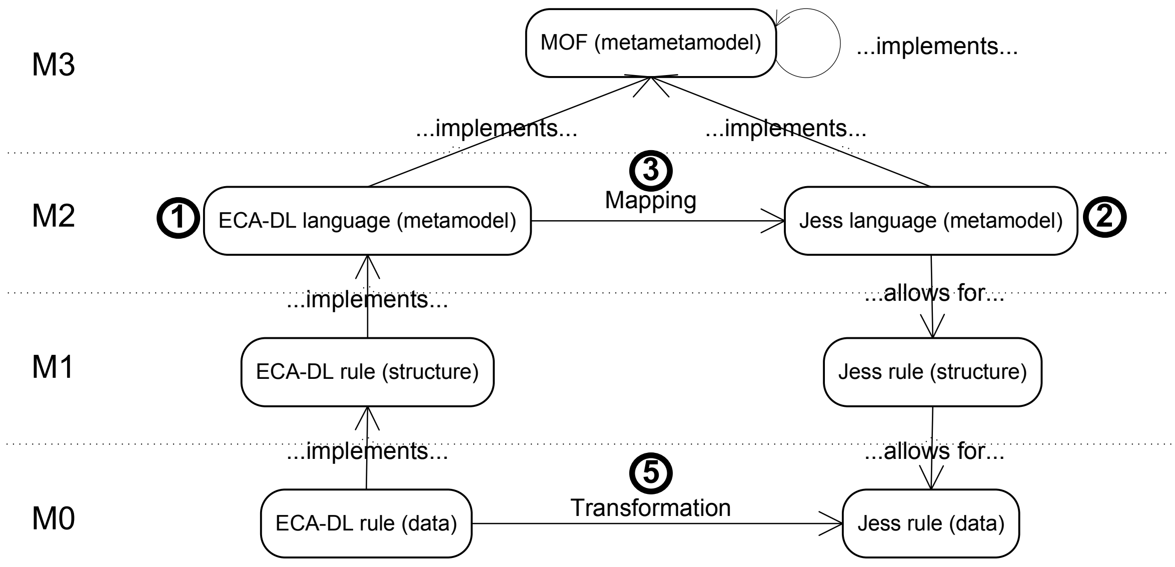


Figure 2: Model hierarchy

In this diagram, there is also a distinction between the structure of a model, and the data in a model. E.g., the structure of a model may define a Person, having a link to an Object. In the data in a model, we then find e.g. the Person “Pete”, and the Object “Computer”. In this project, we do not use the M1 level (structure of a model). Instead, models containing data directly conform to their metamodels. This allows for an easier transformation process, because we do not have to take typing of entities into account (i.e., “Pete” is a Person, or “Computer” is an Object). Also, this is the way our transformation language works by default.

The transformation process itself then consists of two main parts. The first part consists of steps 1, 2 and 3 in Figure 2 and deals with the mapping between ECA-DL and the Jess language. The second part consists of steps 4, 5 and 6 and deals with the transformation of actual rules. Because steps 4 and 6 deal with textual rules, they are not included in Figure 2. Instead, they can be found in the diagram shown in Figure 3.

In steps 1 and 2, we create the ECA-DL and Jess metamodels. These models describe the structure of ECA-DL and Jess rules.

In step 3 we create a mapping between the ECA-DL and Jess metamodels. An analogy to natural language would be the translation from English to Dutch. In English, simple sentences follow the structure `<noun> <verb> <space> <time>`, e.g., “I am at school now”. In Dutch, `<noun> <verb> <time> <space>` is often preferred, e.g., “Ik ben nu op school”. This mapping is specified using a transformation language.

Once steps 1, 2 and 3 have been completed, we can start with the transformation of actual rules. This process is shown in Figure 3.

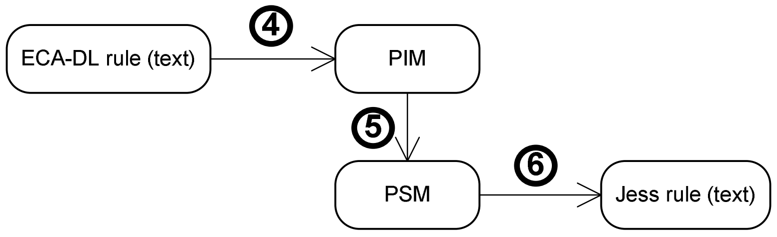


Figure 3: Transformation process

In step 4 we start with the transformation of actual rules. We make a model of an ECA-DL rule, which can, e.g., contain a Person (in the case of the example from Section 2.1 the person Pete) with a link to a Computer (e.g., Pentium IV) called belongsTo. This conforms to the entity Condition from the ECA-DL metamodel. This step is performed using model-to-text transformation. In MDA terminology, we now have our PIM model, because ECA-DL is a platform-independent language.

In step 5 the actual transformation takes place, which is done (semi)-automatically by a transformation tool. In case this transformation is not completely automated, some manual activities are also required in this step.

For the automated part of this model-to-model transformation, the OMG has specified the Queries/Views/Transformations (QVT) standard [16]. Queries can be used to access the source model and extract information from it, Views can be used to focus on specific aspects, and Transformations can be used to take the information from the Queries and produce a target model.

After this step, we now have our PSM model, because Jess is a platform-specific language.

In step 6 the transformed Jess model is transformed back to its textual form, ready for execution in the context-aware application. This step is performed using model-to-text transformation.

In Chapter 6 we present a use case that illustrates this transformation process.

2.4. Development environment

In this project, we have mainly used the Eclipse framework, which supports most of the necessary steps to be taken in the proposed transformation process by means of plugins.

One of the plugins we have used is EMF, the Eclipse Modeling Framework [17]. This plugin provides a framework for "building tools and other applications based on a structured data model". It has support for Ecore, the Eclipse variant of OMG's Meta-Object Facility (MOF), which we have used in this project.

The second plugin we have used is ADT, the ATL Development Tools [18]. This plugin provides an integrated development environment (IDE) for developing ATL transformation specifications and executing transformations. It uses the Atlas Transformation Language [18] (ATL), an early implementation of the QVT standard mentioned above. Instead of UML, ADT uses the so-called Kernel Meta Meta Model (KM3) [19] language to specify metamodels.

Besides Eclipse, we have used the StarUML program [20] to create UML models. StarUML supports a plugin for the automated transformation of UML models to KM3, the language used for specifying metamodels in ADT.

In order to test the outcomes of the transformation, we have used the Jess rule engine [12].

2.5. ATLAS Transformation Language

The ATLAS Transformation Language (ATL) [21, 22] is a model transformation language, developed as an early implementation of the OMG's QVT standard. It is supported by Eclipse's ADT plugin, mentioned above. We present here the basics of ATL, which plays an important role in this project.

ATL is a way for developers to specify how a set of source models should be transformed to create a set of target models. This is realized through rules, which can be either declarative or imperative. Besides rules, an ATL transformation specification can also contain helpers, queries and other constructs.

It is important to note that ATL rules are very different from ECA-DL rules and Jess rules. ATL rules describe how *other* rules, like ECA-DL or Jess rules, should be transformed.

In this project, we use only declarative ATL rules. These can be divided into two types, namely matched rules and lazy rules.

A matched rule is called for every element in the source model that conforms to the header of the rule. It does not need to be called explicitly. A matched rule may look as follows:

```
rule Input2Output
{
    from in_variable : MMInput!Person
    to
      out_variable : MMOutput!Citizen
      (
          name <- in_variable.name,
          nationality <- 'Dutch',
          father <- in_variable.father,
          mother <- in_variable.mother
      )
}
```

When the input model is transformed, this rule is called for every element in the source model that belongs to the class "Person" in the input metamodel "MMInput". Each such element is then transformed to the class "Citizen" in the output metamodel "MMOutput". The `out_variable` is called the target model element.

In this example, the attributes "name" and "nationality", and the references "father" and "mother" are filled in. An attribute is a basic value, belonging to a class. Such values do not need further processing. A reference points to another element, possibly of another class, e.g. by means of a composition link. A reference can be filled in by another target model element.

It is also possible to only have this rule fired when the element in the source model matches certain criteria. These criteria can be mentioned in the from-part of the rule:

```
rule Input2Output
{
    from in_variable : MMInput!Person
            (in_variable.name = 'Pete')
...
```

In this case, only those persons called "Pete" are transformed.

Lazy rules, on the other hand, are not called automatically, but are called explicitly. The syntax of a lazy rule looks much like the syntax of a matched rule, except for the keyword "lazy":

```
lazy rule Input2Output
{
    from in_variable : MMInput!Person
    to
      out_variable : MMOutput!Citizen
      (
          name <- in_variable.name,
          nationality <- 'Dutch',
          father <- in_variable.father,
          mother <- in_variable.mother
      )
}
```

This rule works the same as the matched rule above, but it has to be called explicitly. This can be done in another rule as follows:

```
...
      to
      out_variable : MMOutput!City
      (
          citizen <- thisModule.Input2Output(in_variable.person),
...
```

The main reason to use lazy rules is to transform certain parts of the source model multiple times. A matched rule is called only once per element, but a lazy rule can be called as many times as necessary, because it is called explicitly.

In Chapter 5 we present the transformation specification using ATL.

3. Metamodels

In this chapter we present the metamodels of ECA-DL and Jess and we show how they have been constructed.

We have first specified these metamodels using UML. This allowed us to make a graphical representation of the metamodels for easier understanding. When a metamodel had to be used in the ATL environment, it was transformed to the textual KM3 format. The KM3 versions of our metamodels can be found in Appendix A.

3.1. ECA-DL metamodel

Because the complete ECA-DL metamodel is rather extensive, we have decided to introduce the constructs in three steps. The first, very simple metamodel shows only the main ECA-DL parts. The second metamodel shows related classes, which show what the ECA-DL parts consist of. The third and final metamodels also shows the extended constructs Lifetime, Select and Scope.

3.1.1. Event, Condition and Action

The main elements of an ECA-DL rule are Event, Condition and Action. This can be modeled as shown in Figure 4.

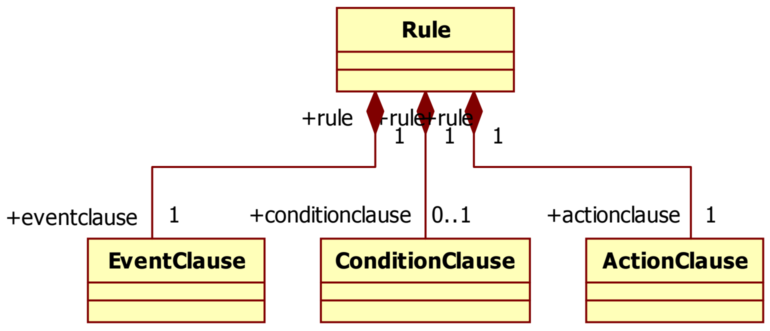


Figure 4: The main elements of an ECA-DL rule

Figure 4 shows that a Rule is composed of the three parts EventClause, ConditionClause and ActionClause. By using composition, we indicate that the parts cannot exist outside of a Rule.

The cardinalities indicate that an ECA-DL rule always has an EventClause and an ActionClause, and may also have a ConditionClause.

Finally, the navigability is bidirectional, which indicates that the parts are accessible from the Rule and the other way around (i.e., a part can access its rule). When, e.g., the EventClause is mapped to Jess, the transformation engine should have access to the Rule in order to determine if the rule contains a ConditionClause or not.

3.1.2. Basic ECA-DL metamodel

Figure 5 shows the model in more detail by showing related classes and how all classes relate to each other.

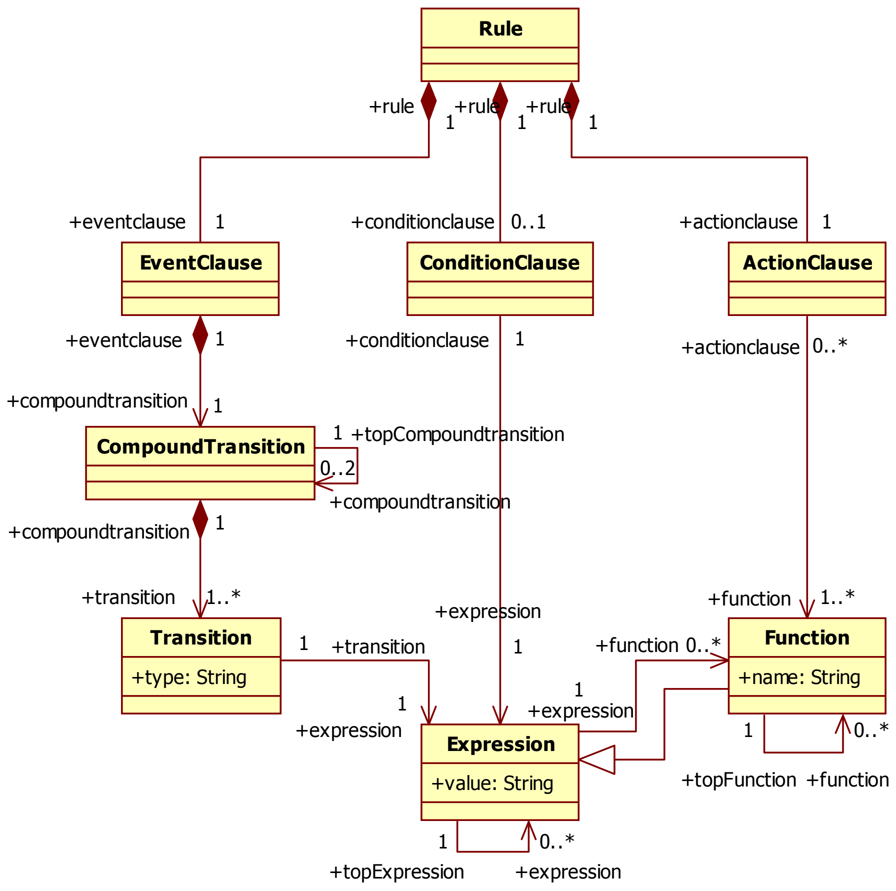


Figure 5: The basic ECA-DL metamodel

By using composition we indicate that a Transition cannot exist outside of a CompoundTransition, which in its turn cannot exist outside of an EventClause. An EventClause can contain only one CompoundTransition. As the name implies, this CompoundTransition can contain other CompoundTransitions, and ultimately Transitions themselves (hence the one-to-many cardinality to Transition). CompoundTransitions are built using the AND and OR operators. These are binary operators, hence the cardinality of at most 2. The logical NOT operator is not allowed here, as this would model the absence of a transition. This operator is only allowed in expressions.

Figure 5 shows that, unlike Expressions, Transitions cannot be nested. A Transition *describes* a change, but cannot change itself. A Transition describing another Transition therefore does not make sense.

Similarly to a Transition, a ConditionClause can have only one expression. Expression has an association to itself, indicating that it can be composed of multiple expressions, e.g., by means of the AND, OR and NOT operators. In this way, multiple conditions can be checked. It can also contain functions.

A special case of Expression is Function. Because a Function is defined outside a Rule, it can be used in several ActionClauses. An ActionClause can also contain several Functions. As in most programming languages, functions take a number of arguments (zero or more), which are expressions. Similarly to Expression, Function has a link to itself, so a Function can also contain other Functions. This link acts as a refinement.

The model above allows the following rule:

```
Upon EnterTrue(PeteHome OR MaryHome) AND EnterFalse(LightIsBurning)
Do TurnOnLight()
```

In this example, Transitions only contain Expressions, but not other Transitions. All the Transitions together form a CompoundTransition. Figure 6 shows how this example rule relates to the metamodel.

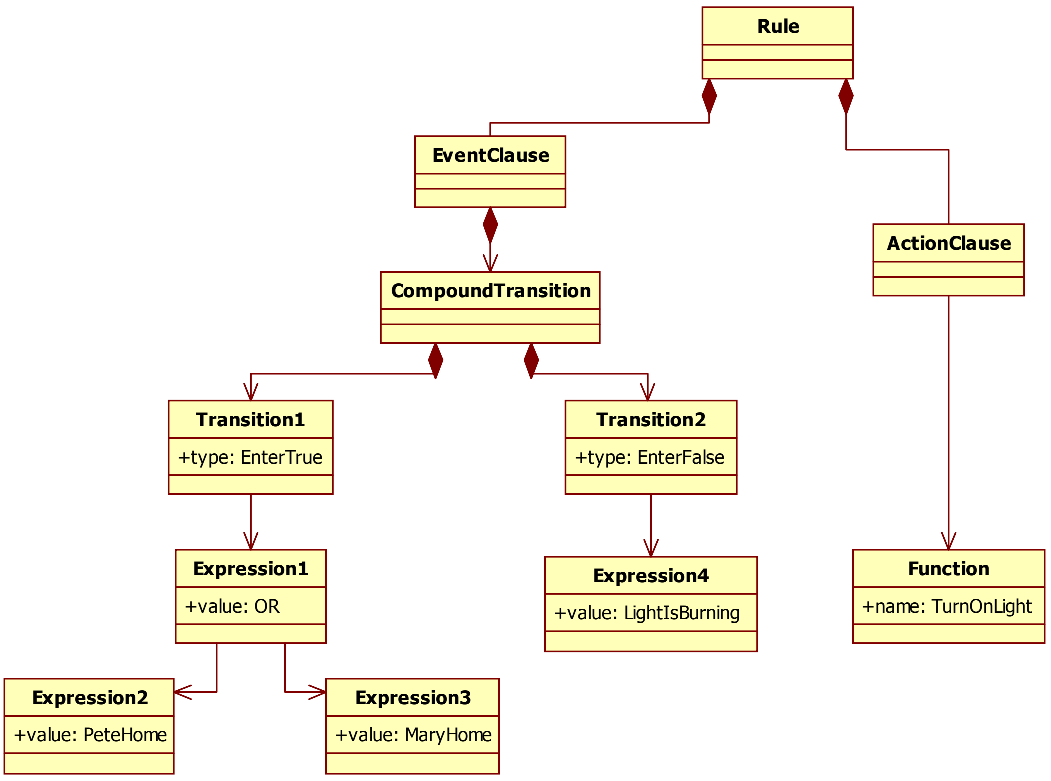


Figure 6: An example ECA-DL model

However, the following would not be possible:

```
EnterTrue(PeteHome OR EnterTrue(Notify(Mary)))
```

In this example, one Transition contains another Transition. The metamodel does not allow this.

3.1.3. Extended ECA-DL metamodel

Introducing the Lifetime, Scope and Select constructs of ECA-DL, we get the final metamodel of ECA-DL, as shown in Figure 7.

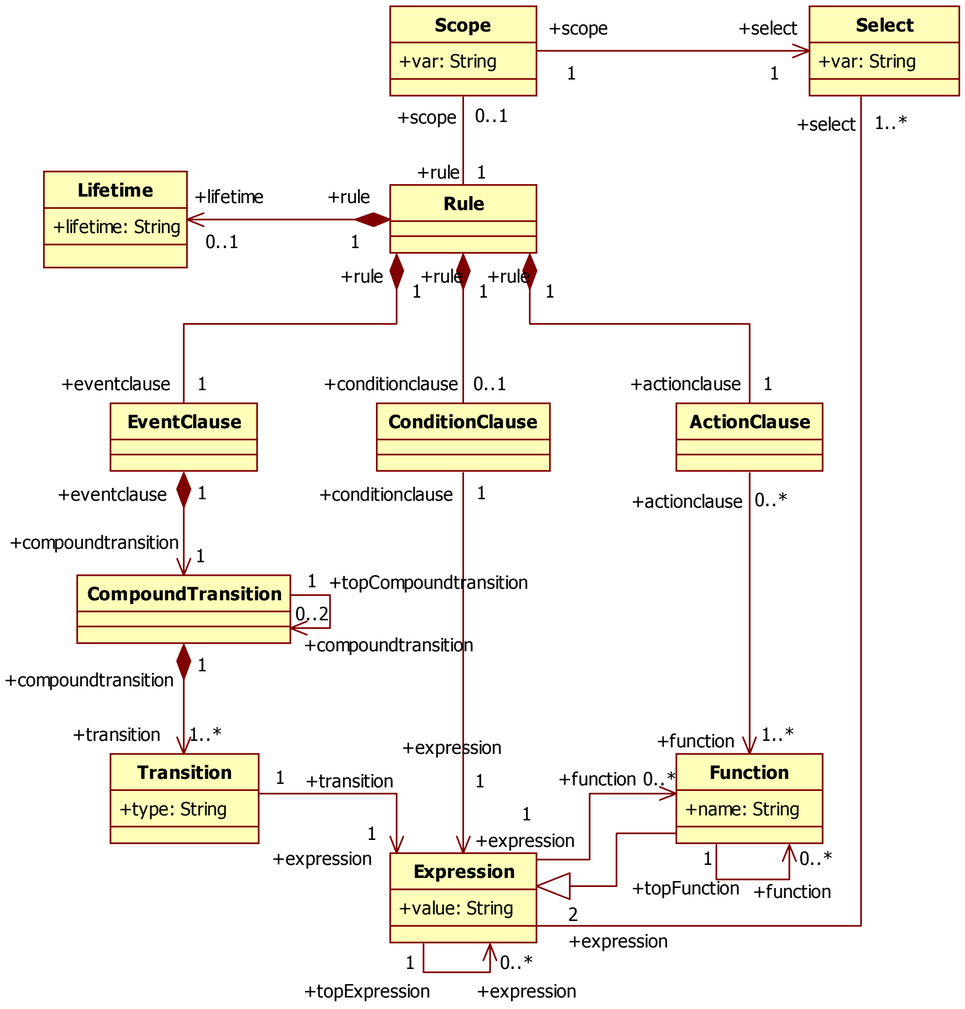


Figure 7: The extended ECA-DL metamodel

A rule can possibly be surrounded by a Scope block, which also contains a Select expression and a variable. The bidirectional navigability indicates that a Rule may access the Scope it is in. This is necessary when the Scope's attribute `var` is used in a Rule.

A Select expression also contains such an attribute, as well as two expressions for the collection of entities and a condition that has to be fulfilled for each of these entities. An Expression can contain a number of Selects. This is comparable to CompoundTransition, which also has a one-to-many link to Transition.

Finally, a Rule may have a Lifetime. This Lifetime indicates when the rule should be active and when it should be inactive.

3.2. Jess metamodel

Because the complete Jess metamodel is simpler than the complete ECA-DL metamodel, we need only two steps to introduce it. As in Section 3.1.1, the first metamodel shows only the main Jess parts. The second and final metamodel shows related classes and shows how all classes relate to each other.

3.2.1. Left-Hand Side and Right-Hand Side

The basic structure of a Jess rule can be modeled as shown in Figure 8.

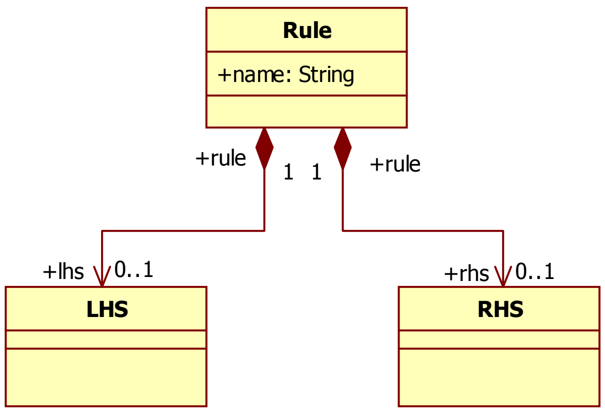


Figure 8: The main elements of a Jess rule

We see here a clear distinction between ECA-DL rules and Jess rules. Whereas an ECA-DL rule contains three parts, a Jess rule consists of only two. Also, these parts are optional.

In practice this means that a rule without an LHS never is fired, and rules without an RHS do not do anything. However, on the syntactic level it is possible to construct such rules, so the metamodel should reflect this.

Unlike the ECA-DL metamodel, navigability is only towards the parts, because during the transformation of either part, the transformation engine does not need access to the other.

3.2.2. Jess metamodel

The diagram shown in Figure 9 shows the metamodel in more detail. The figure shows related classes and how all classes relate to each other. This is also the final Jess metamodel.

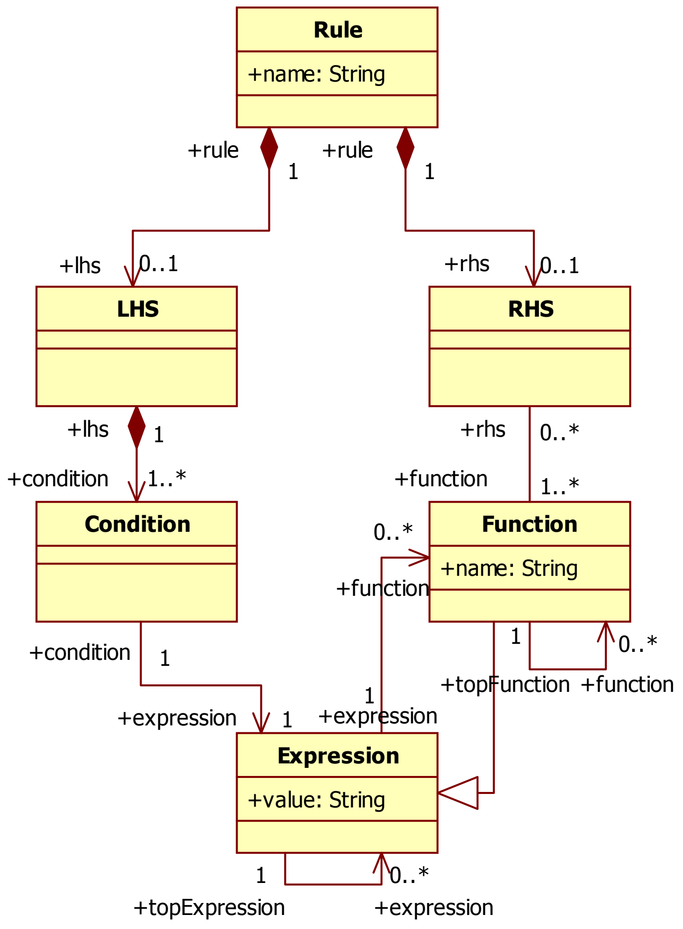


Figure 9: The Jess metamodel

Although an LHS can contain several Conditions, internally these are treated as one compound statement using the AND operator. In this way, it is rather similar to the Condition in ECA-DL. The Expression part is equal in both metamodels.

The RHS is even more like its ECA-DL counterpart. As in ECA-DL, Functions can be defined outside rules and used in several RHS's. A Function can contain zero or more Expressions and zero or more other Functions, just like Expression.

The metamodel above allows the following rule:

```
(defrule turn-lights-on
  (or (Location Pete Home) (Location Mary Home))
  =>
  (turnOnLights)
)
```

Figure 10 shows how this example rule relates to the metamodel.

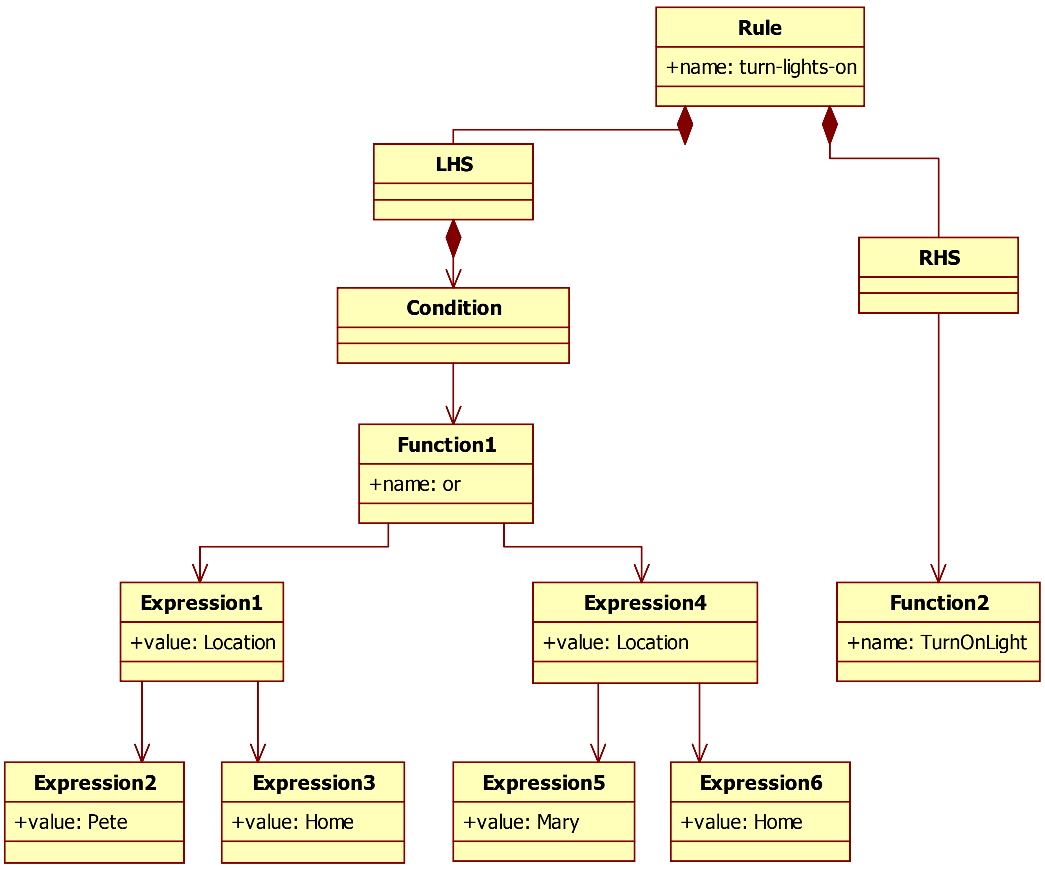


Figure 10: An example Jess model

4. Mapping ECA-DL to Jess

In this chapter we discuss how ECA-DL constructs can be mapped to Jess constructs and we evaluate several alternatives for doing so. This mapping is specified in an informal way, using textual rules instead of models. Chapter 5 is devoted to specifying this mapping in a more systematic way using the ATLAS Transformation Language (ATL).

4.1. Basic ECA-DL syntax

Because the desired transformation is one-way from ECA-DL to Jess, our focus is completely on mapping ECA-DL constructs to Jess constructs. Initially, we restrict ourselves to just the basic ECA-DL syntax. Later on, we deal with ECA-DL's additional constructs.

4.1.1. Rules without conditions

If an ECA-DL rule only contains an Event and an Action part, then the mapping to Jess is very straightforward. Take the following ECA-DL rule:

```
Upon EnterTrue(inRoom(Mary))
Do startWorking()
```

The Event part states that the rule should be triggered when Mary is in the room. This part is mapped to Jess' LHS. The Action part states that when the event occurs, she should start working. This part is mapped to Jess' RHS. In Jess, this solution would be represented as follows:

```
(defrule rule-1
  (inRoom Mary)
  =>
  (startWorking)
)
```

4.1.2. Rules with conditions

When an ECA-DL rule also contains a Condition part, the mapping is more complex. In an ECA-DL rule, the Event part describes when the rule should be fired and contains one or more state transitions. A possible Condition part specifies additional constraints that should hold before the rule is fired. This implies that an event is thus very different from a condition. Whereas a condition just describes constraints that should hold, an event actually models a change. This is clarified by the following examples:

- `EnterTrue(inOffice)`
  This event describes a *change*, which happens at a specific *point* in time;

- `inOffice`
  This condition describes a *situation*, which holds over a *period* of time.

Jess does not make such a distinction. In Jess, both the Event and Condition part are represented by Conditions in the LHS. The RHS then gets executed as soon as all the conditions in the LHS evaluate to true. A naïve way to map the Event and Conditions part to Jess would be to simply place both in the LHS. Take the following example:

When Mary enters the room, and Pete is already there, act surprised.

In ECA-DL, this can be represented as follows:

```
Upon EnterTrue(inRoom(Mary))
When inRoom(Pete)
Do actSurprised()
```

Because Jess rules only get fired as soon as all conditions hold, the `EnterTrue` function is implicit. The Jess counterpart would be:

```
(defrule rule-1
  (inRoom Mary)
  (inRoom Pete)
  =>
  (actSurprised)
)
```

Although this solution is quite straightforward, it does not behave correctly according to the ECA-DL semantics. Suppose Pete enters the room first, and then Mary enters the room. Both conditions now hold, and the rule is fired, as it should. However, in case Mary enters first, and then Pete enters, the Jess rule still is fired, contrary to the ECA-DL semantics. Jess behaves in this way because it makes no distinction between the different conditions in the LHS.

Instead, we want Jess to only fire the rule when the event occurs (Mary enters the room) and the condition *already* holds (Pete is in the room). A possible solution is to introduce an auxiliary fact, which exists as long as the condition holds, and is removed when the condition no longer holds. Only when this fact exists, can the rule be fired. This ensures that at the moment the event occurs, the conditions hold. In order for this to work, we need three rules instead of one:

1. Rule 1 is fired as soon as the condition holds. This adds the fact `fact-cond-holds`.

2. Rule 2 is fired when the event occurs and the fact mentioned above holds. This triggers the actual actions.

3. Rule 3 is fired as soon as the condition does not hold anymore. This removes the fact mentioned above, and thus prevents rule 2 from firing.

However, this solution is not yet complete. Suppose Mary enters the room, and then Pete enters the room. Rule 1 is fired and it adds the fact `fact-cond-holds`. Rule 2 also immediately is fired, because Mary is also in the room. In order to solve this problem, we need to modify rule 1 to include the negation of the event. This negation models the absence of a fact instead of its existence. This way, rule 1 only is fired when Pete is in the room and Mary is not yet in the room.

Finally, we have to give priority to rule 3 to always fire before rule 2 when both can be fired. This is needed to ensure that rule 2 does not get fired directly after the condition does not hold anymore.

In Jess, this first solution would be represented as follows:

```
(defrule rule-1
  (inRoom Pete)          ; condition
  (not (inRoom Mary))    ; negation of event
  =>
  (assert (fact-cond-holds))
)

(defrule rule-2
  (inRoom Mary)          ; event
  (fact-cond-holds)      ; fact from above
  =>
  (actSurprised)
)

(defrule rule-3
  (declare (salience 1)) ; give priority to this rule
  (not (inRoom Pete))    ; negation of condition
  =>
  (retract-string "(fact-cond-holds)")
)
```

The main drawback of this solution is the need for an auxiliary fact. Such a fact may interfere with already existing facts, causing a name clash.

It is possible to drastically reduce the chance of auxiliary facts interfering with already existing facts by introducing a separate namespace for these facts. We call this namespace "ECADL2Jess". The following code is placed at the top of the first solution:

```
(defmodule ECADL2Jess)
(set-current-module MAIN)  ; switch back to default module
(assert (ECADL2Jess::fact))
```

It is also possible to remove the need for an auxiliary fact. Instead of using an auxiliary fact as a link between rule 1 and rule 2, rule 1 now *creates* rule 2. Without the condition holding, rule 2 simply does not exist. The mapping is then as follows:

1. Rule 1 is fired as soon as the condition holds. This adds rule 2 below.

2. Rule 2 is fired when the event occurs. This triggers the actual actions.

3. Rule 3 is fired as soon as the condition does not hold anymore. This removes rule 2.

In Jess, this second solution would be represented as follows:

```
(defrule rule-1
  (inRoom Pete)          ; condition
  (not (inRoom Mary))    ; negation of event
  =>
  (build (str-cat "(defrule rule-2
                   (inRoom Mary)
                   =>
                   (actSurprised)
                   )"
         ))
)
```

```
(defrule rule-3
  (declare (salience 1)) ; give priority to this rule
  (not (inRoom Pete))    ; negation of condition
  =>
  (undefrule rule-2)
)
```

However, this solution brings an even greater drawback. In the rule above, we notice the use of the `(build)` function. This function takes its argument as a string and then evaluates it as if it were normal code. This is usually seen as bad programming, because the programmer has little control over the code that is executed by such a function.

Furthermore, in this case the rule created by this `(build)` function cannot be checked with respect to the metamodel for conformance, because it is treated as plain text. This also means that we have Jess code in the mapping itself, as an argument to the `(build)` function.

Because we believe the drawback of the second solution is more severe than the drawback of the first solution, we choose to use the first solution for our mapping.

We do not elaborate on the mapping of the Action part from ECA-DL to Jess, because it is a trivial one-to-one mapping.

4.1.3. Transitions

In Section 4.1.1 we have seen that transition functions, as used in ECA-DL, are implicit in Jess. The ECA-DL event `EnterTrue(inRoom(Mary))`, transformed to Jess, is thus simply `(inRoom Mary)`.

However, ECA-DL supports more transitions than just `EnterTrue`. Because a state can have three possible values (true, false or unknown), there are six different transitions. These can be denoted using thirteen different functions (e.g., `Changed()`, `FalseToTrue()` or `ExitUnknown()`) [4].

Unfortunately, not all of these functions can be mapped to Jess. In Jess, there is no "unknown" value, since conditions always evaluate to either true or false. Because of this, there is also no distinction anymore between `EnterTrue` and `FalseToTrue`, since a transition to state true must have come from state false. The functions that can be mapped to Jess are thus: `EnterTrue()`, `FalseToTrue()`, `EnterFalse()`, `TrueToFalse()` and `Changed()`. Other transition functions are not supported.

These functions can be mapped to Jess as follows:

- `EnterTrue()` and `FalseToTrue()`
  These functions are simply ignored. `EnterTrue(inRoom(Mary))` and `FalseToTrue(inRoom(Mary))` thus both become `(inRoom Mary)`;

- `EnterFalse()` and `TrueToFalse()`
  These functions are replaced by the negation of the event. `EnterFalse(inRoom(Mary))` and `TrueToFalse(inRoom(Mary))` thus both become `(not (inRoom Mary))`.
  In order for Jess to understand the negation of an event, or more specifically, the absence of a fact, it needs the initial fact. This fact is placed in working memory by issuing the `(reset)` command. Because issuing this command is seen as good practice, it is placed before any generated Jess rule;

- `Changed()`
  This function is replaced by the combination (by means of the OR operator) of both the event and the negation of the event, i.e., when the event or the negation of the event happens, the rule is fired. `Changed(inRoom(Mary))` thus becomes `(or (inRoom Mary) (not (inRoom Mary)))`.
  Although logically, this statement is always true, in Jess, rules are only triggered for those facts that have *changed*. Thus, when the fact `(inRoom Mary)` changes, only then is the rule always triggered. This is exactly the intended behavior.

4.1.4. Compound transitions and conditions

Until now, we have only worked with single events and conditions, like `inRoom(Pete)`. However, both ECA-DL and Jess also support compound transitions and conditions. To illustrate this we change the example from Section 4.1.2 as follows:

```
Upon EnterTrue(inRoom(Mary)) AND EnterTrue(inRoom(Suze))
When inRoom(Pete) AND inRoom(Carl)
Do actSurprised()
```

In ECA-DL, this example is interpreted as follows:

The event occurs when both Mary and Suze have entered the room, i.e., before the event, either or both were not yet in the room. It does not matter who enters the room first. The condition states that the rule is fired only when Pete and Carl are already in the room the moment the event takes place. This may be counterintuitive, as can be seen from the following scenario:

- Pete enters the room;
- Mary enters the room;
- Carl enters the room;
- Suze enters the room. The event occurs (both Mary and Suze are in the room). The condition holds (both Pete and Carl are in the room). The rule is fired and the action gets triggered.

Intuitively, one may expect the rule to only fire when first Pete and Carl enter the room, and after that Mary and Suze. However, ECA-DL states that the condition should only hold the moment the event occurs, not during every state transition in the event part. Fortunately, this easily translates to Jess. In Jess, this example would be represented as follows:

```
(defrule rule-1
  (and (inRoom Pete) (inRoom Carl))          ; condition
  (not (and (inRoom Mary) (inRoom Suze)))    ; negation of event
  =>
  (assert (fact-cond-holds)
)

(defrule rule-2
  (and (inRoom Mary) (inRoom Suze))          ; event
  (fact-cond-holds)                          ; fact from above
  =>
  (actSurprised)
)
```

```
(defrule rule-3
  (declare (salience 1))                    ; give priority
  (not (and (inRoom Pete) (inRoom Carl)))   ; negation of condition
  =>
  (retract-string “(fact-cond-holds)”)
)
```

The mappings of the event and condition are quite straightforward, as they are the same as in Section 4.1.1 and Section 4.1.2. We show here that the mapping of the *negations* of the events and conditions are also correct.

The negation of the event states: “if not (Mary is in the room and Suze is in the room)”. This is equivalent to: “if Mary is not in the room, or Suze is not in the room”, or “if either Mary, or Suze, or both are not in the room”. If this is the case, then either Mary, Suze or both still have to enter the room, and the event still has to happen.

The reasoning for the negation of the condition is similar. If either Pete or Carl is not in the room (yet or anymore), then the condition evaluates to false, and the action can no longer get triggered.

There is however one constraint we have to keep in mind. Jess supports multiple conditions in its LHS, but our Jess metamodel states that these conditions, unlike ECA-DL’s transitions, cannot be nested. In our metamodel, nested Jess Conditions are represented by Expressions instead.

We can overcome this problem by always mapping Transitions and Conditions to Expressions. These Expressions are then placed in a single Jess Condition. This is possible, because:

- The ECA-DL Transitions and Conditions are still mapped to Jess Conditions as in Section 4.1.1 and Section 4.1.2;
- Section 4.1.3 shows that Transitions and Conditions are already mapped to Expressions;
- Expressions in Jess can be nested.

4.2. Additional constructs

4.2.1. Lifetime

ECA-DL rules can have a lifetime, indicating when the rule should be active. There are six different lifetimes:

- `always`
  This lifetime indicates that the rule is always active. Because Jess rules are always active by default, this lifetime can be ignored in the transformation.
- `once`
  This lifetime indicates that the rule should be deactivated when it has been triggered once. In Jess, there are two ways to implement this:

- The Jess rule can remove itself:

  ```
  (reset)
  (defrule rule-1
    (inRoom Mary)        ; event
    =>
    (startWorking)       ; action
    (undefrule rule-1)   ; “once”
  )
  ```

- The Jess rule can assert a new fact in the action part which prevents it from firing again:

  ```
  (defmodule ECADL2Jess)
  (set-current-module MAIN)
  (defrule rule-1
    (inRoom Mary)                       ; event
    (not (ECADL2Jess::rule-1-fired))    ; “once”
    =>
    (startWorking)                      ; action
    (assert (ECADL2Jess::rule-1-fired)) ; “once”
  )
  ```

In the second solution, the rule stays in the knowledge base and can later be activated again. Because of this possibility, we choose to use the second solution for our mapping.

- `<n>`
  This lifetime indicates that the rule should be deactivated when it has been triggered <n> times. In Jess, this can be implemented by introducing a counter variable. When the counter reaches zero, the rule can no longer be triggered. In the example below, we use ‘5’ as our value for <n>:

  ```
  (bind ?*rule-1-count* 5)
  (defrule rule-1
    (inRoom Mary)
    (test (> ?*rule-1-count* 0))
    =>
    (startWorking)
    (-- ?*rule-1-count*)
  )
  ```

  For our counter we use the variable `?*rule-1-count*`. The asterisks around the name of the variable indicate that this is a global variable. A normal variable (e.g. `?rule-1-count`) cannot be used here, because it is not visible within the rule.

- `from <start> to <end>`
  Although Jess has a `(time)` function, which returns the number of seconds since 12:00 AM, Jan 1, 1970 (Unix epoch), it cannot be used in a rule. In order for a rule to start firing as soon as the lifetime permits this, the time needs to be checked in the LHS of the rule.

  However, the `(time)` function is never evaluated, so the rule does not match. The reason for this is the way Jess works internally. It would be too costly to evaluate the `(time)` function every time Jess searches for matches between facts and rules.

  Even if this `(time)` function had worked properly, then still only timestamps (seconds since Unix epoch) would have been supported, because Jess has no method of working with textual dates, like “September 23rd, 1997”.

It is thus not possible to map this lifetime to Jess, so this lifetime is not supported.

- `to <end>`
  The same reasoning as above applies.

- `frequency <n> times per <period>`
  The same reasoning as above applies.

4.2.2. Scope and Select

The Scope construct of ECA-DL allows multiple, similar rules to be defined at once. This is called “parameterization”.

The Select construct is used to select a number of entities from a collection. The metamodel allows a Select to be used wherever an Expression can be used, but ECA-DL only allows a Select construct in a Scope construct or in the Action part of a rule.

Because Select constructs are often used in Scope constructs, we do not discuss them separately. After that, we discuss the use of the Select construct in the Action part of a rule.

Although Jess does not support parameterization as ECA-DL does, it is relatively easy to develop a mapping that obtains the same result. Jess rules are fired every time one of the parts of the LHS changes, and as many times as there are facts that match the rule. Take the following example:

When a manager enters his office, turn on his computer.

In ECA-DL, this can be represented as follows:

```
Scope(Select(Employees.*, Employee, Employee.isManager); emp)
{
  Upon EnterTrue(inOffice(emp))
  Do TurnOn(emp.computer)
}
```

For example, if manager1 and manager2 enter their offices, the rule is fired for both. In Jess, we can write this as follows:

```
(defrule rule-1
  (Employees ?emp ?function&:(eq ?function “manager”))
  (inOffice ?emp)
  =>
  (turnOn computer(?emp))
)
```

On the first line of the LHS we see the references to `Employees` and `Employee.isManager` from the Select construct. Here we also see the reference to `emp`, the variable from the Scope construct. This line makes sure that the rule is fired when an Employee enters the office and this Employee is a manager.

Thus, the Scope and Select constructs are mapped to a single Condition in Jess. It is important to mention that this Condition should be the first one, so that the variable `?emp` introduced here is available to all the other Conditions in the Jess rule, as in the case of the ECA-DL rule.

This mapping introduces a problem when used in conjunction with a Lifetime. In ECA-DL, a Lifetime is bound to a single Rule. When a Scope is placed around such a Rule, each Rule still has its own Lifetime. However, when mapped to Jess, Scope and Rule merge, so that the Lifetime now works over the single created rule, instead of over the original set of rules.

Besides this, the Select construct can also occur in the Action part of an ECA-DL rule. In this case it is simply a list of entities that can be passed to a function. Because a Select expression works with facts, we need a Jess function that can return a number of facts that satisfy the given condition. We can use the `(defquery)` construct for this. Take the following example:

When Pete enters the office, alert all his colleagues.

In ECA-DL, this can be represented as follows:

```
Upon EnterTrue(inOffice(Pete))
Do Notify(Select(Employees.*, Emp, Emp.isColleagueOf(Pete)),
          "Pete entered the building.")
```

Using the `(defquery)` construct, we get the following Jess code:

```
(defquery query-1
  (declare variables ?Emp)
  (Employees ?Emp)
  (Colleague ?Emp ?Col)
)
(defrule rule-1
  (InOffice Pete)
  =>
  (Notify (run-query* query-1 Pete),
          "Pete entered the building.")
)
```

The `(defquery)` construct creates a query function that returns all the facts that satisfy the conditions, in this case, all the colleagues of Pete. This function is used in the RHS of the rule.

5. Transformation specification

In this chapter we present how the mappings from Chapter 4 can be formalized so that they can be used in a transformation tool. While in Chapter 4 we focused on mappings using text, ECA-DL and Jess code, in this chapter we present these mappings using the ATLAS Transformation Language (ATL).

We also present how these mappings relate to the metamodels from Chapter 3.

5.1. Basic mapping

In Chapter 4 we introduced two basic mappings from ECA-DL to Jess. One does not consider the Condition part of the ECA-DL rule, yielding a rather simple mapping. This mapping is shown in Section 4.1.1. The other does include the Condition part, yielding a rather complex mapping. This mapping is shown in Section 4.1.2.

These two mappings are more important than the other mapping, because the other mappings are nested in these two mappings. Therefore, first one of these two mappings has to be performed for the others to be meaningful.

5.1.1. Event and Action only

When a rule does not contain a Condition part, then the mapping is very straightforward; one ECA-DL rule is mapped to one Jess rule. This is shown in Figure 11.

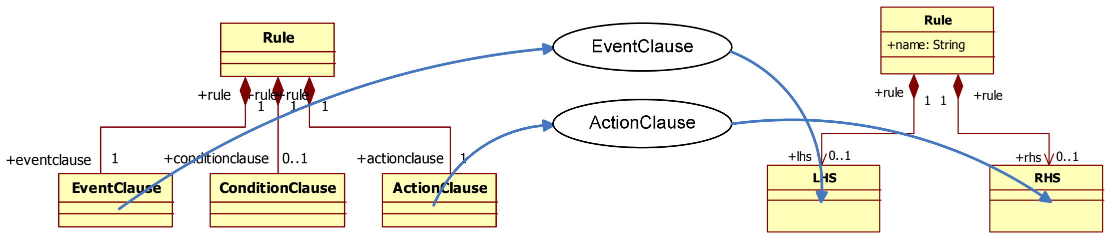


Figure 11: Mapping for Event & Action

Below we present the specification of this mapping in ATL.

The header below defines that only those rules that do not contain a Condition part are transformed using this transformation rule. Those rules that do contain a Condition part are transformed in another transformation rule.

```
rule Rule2RuleSimple
{
    from
        ecadl_rule : MMECADL!Rule
            (not ecadl_rule.hasCondition)
```

The code below defines the basic structure of the Jess rule. It creates the name of the rule, and the LHS and RHS. The LHS and RHS are assigned a target model element generated by the current transformation rule. In the case of `lhs`, this target model element is `jess_lhs`.

```
        jess_rule : MMJess!Rule
        (
            name <- 'rule-1',
            lhs <- jess_lhs,
            rhs <- jess_rhs
        ),
```

In the target model element `jess_lhs` we first make use of a collection. Because collections can contain multiple elements, it is possible to assign multiple children to a reference. In this case, we make use of the `Sequence{}` construct, a type of collection. A Sequence is an ordered set of elements, in which duplicates are allowed. In the target model element below, the Sequence possibly contains one Condition representing the Scope of the ECA-DL rule, and it contains one or more Conditions representing the Transitions of the ECA-DL rule.

Because the Condition generated by the Scope of the ECA-DL rule should be the first Condition in the LHS of the Jess rule, it is placed before the other Conditions.

The target model element below is also the first that explicitly calls a lazy rule. This lazy rule, called `Scope2Condition`, transforms the Scope part of an ECA-DL rule to a Jess Condition. This lazy rule is presented later on. Below, the use of lazy rules is necessary, because the Scope part is used in other transformation rules as well.

```
        jess_lhs : MMJess!LHS
        (
            condition <- Sequence
                  {
                  -- Transform Scope.
                  if not ecadl_rule.scope.oclIsUndefined() then

thisModule.Scope2Condition(ecadl_rule.scope)
                      else
                          Sequence{}
                  endif,
                  -- Transform Conditions.
                  jess_condition
                  }
        ),
        jess_condition : MMJess!Condition
        (
            expression <-
thisModule.CompoundTransition2Expression(ecadl_rule.eventclause.compo
undtransition)
        ),
```

In some cases, the element to be copied is a collection. In this case, the ActionClause from the ECA-DL rule can contain multiple functions. Instead of passing the entire collection to a lazy rule, we call the transformation rule for each element `e`, and then `collect` all the results.

```
        jess_rhs : MMJess!RHS
        (
            function <- ecadl_rule.actionclause.function-
>collect(e|thisModule.Function2Function(e))
        )
```

Finally, we introduce the lifetime. The code below calls two helper functions that add the necessary code to the LHS and RHS. These functions are presented later on.

```
    do
    {
        -- Introduce Lifetime.
        jess_lhs.condition <-
ecadl_rule.extendLifetimeLHS(jess_lhs.condition);
        jess_rhs.function <-
ecadl_rule.extendLifetimeRHS(jess_rhs.function);
    }
```

5.1.2. Event, Condition and Action

When an ECA-DL rule does contain a Condition part, the rule is mapped to three Jess rules. This mapping relates to the metamodels as shown in Figure 12

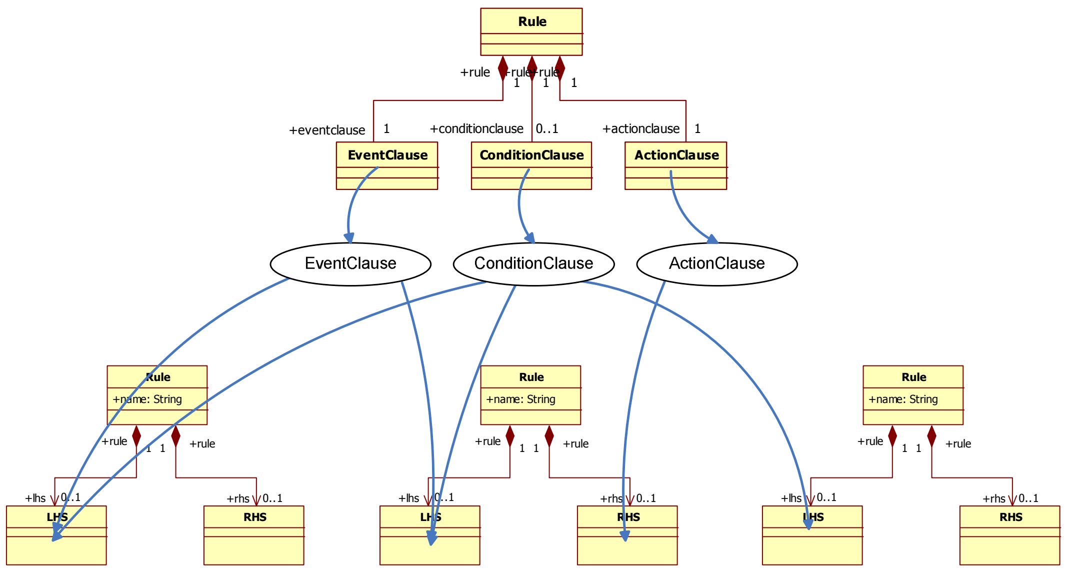


Figure 12: Mapping for Event, Condition & Action

The arrows here indicate which part of the ECA-DL rule is mapped to which part of the Jess rules. Figure 12 shows that the original ECA-DL rule is mapped to three Jess rules. Both the Event and Condition part are mapped to several LHS's. Those parts in Jess not linked to the ECA-DL rule are filled in by the transformation process.

The specification of this mapping in ATL is rather long, because it has to take several properties of the mapping into account. Furthermore, it only contains constructs that have been explained before. Therefore, we do not elaborate on this mapping here. The complete mapping can be found in Appendix C.

5.2. Detailed mapping

Below these basic mappings, there are a number of more detailed mappings. These mappings relate to the metamodels as shown in Figure 13.

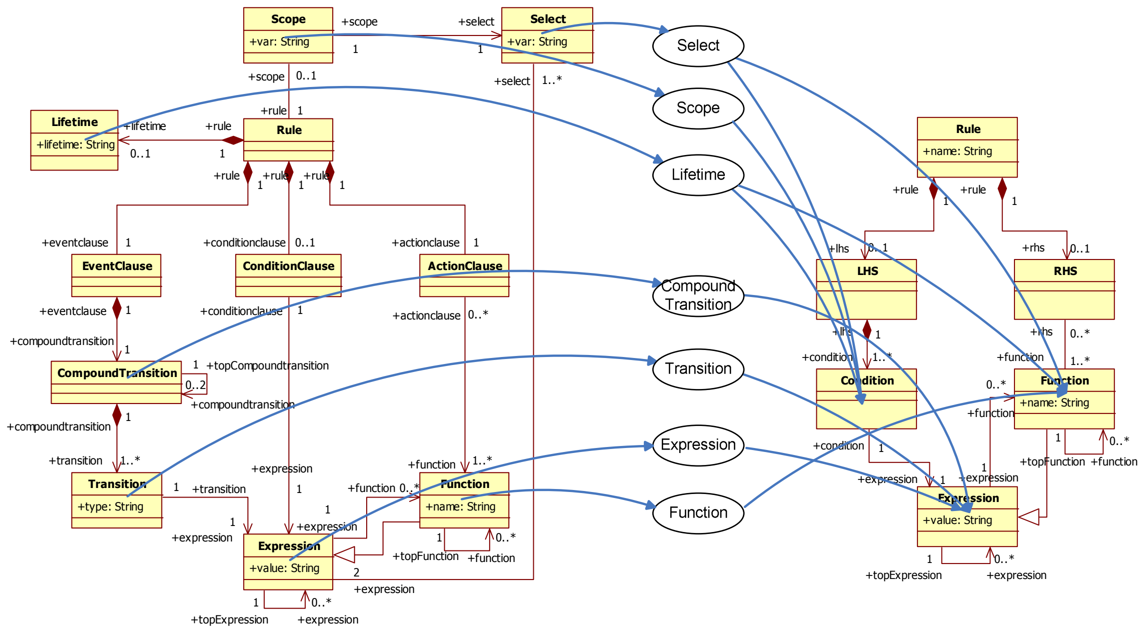


Figure 13: Mappings for other constructs

The diagram in Figure 13 is based on the mapping from Section 5.1.1. The diagram based on the mapping from Section 5.1.2 would be similar. For example, because a CompoundTransition is part of an EventClause, and the EventClause is mapped to two LHS's, the CompoundTransition is then mapped to the Condition parts of these two LHS's. However, this more extensive diagram would become unreadable, so we constrain ourselves to this one. The mappings represented in Figure 13 are:

- Select to Condition and Function, denoted by the Select ellipse;
- Scope to Condition, denoted by the Scope ellipse;
- Lifetime to Condition and Function, denoted by the Lifetime ellipse;
- CompoundTransition to Expression, denoted by the CompoundTransition ellipse;
- Transition to Expression, denoted by the Transition ellipse;
- Expression to Expression, denoted by the Expression ellipse;
- Function to Function, denoted by the Function ellipse.

We discuss in separate sections those mappings that are not too trivial. The complete mapping can be found in Appendix C.

5.2.1. Scope and Select to Condition

The lazy rule `Scope2Condition` can be called either once or three times. If the ECA-DL rule contains no Condition part, then the Scope part needs to be mapped to the LHS of a single Jess rule. If the ECA-DL rule does contain a Condition part, then the Scope part needs to be mapped to the LHS's of all three Jess rules.

The following block of code is an imperative block of code, which can be used to declare variables. In this block, we declare the variable `expressions`. This variable contains the transformed versions of the Expressions belonging to the Select part. The Select part, in its turn, belongs to the Scope part.

```
    using
    {
        expressions : MMJess!Expression =
ecadl_scope.select.expression-
>collect(e|thisModule.Expression2Expression(e));
    }
```

In order to access an element in the collection above, we use the `at()` function. This function returns an element at a certain position in a collection. The Expression part below contains the first element of the collection, then the attribute `var` of Scope, and lastly the second element of the collection.

```
    expression <- Sequence
        {
            expressions->at(1),
            jess_expression_var,
            expressions->at(2)
        }
 ),
 jess_expression_var : MMJess!Expression
 (
     value <- ecadl_scope.var
 )
```

5.2.2. Select to Function

When used in the Action part of an ECA-DL rule, the Select construct is mapped to a Jess query and a Jess function, calling this query. Because it is possible to use the Select construct multiple times, we may need multiple Jess queries, each with their own identifier. In order to ensure each query has a different name, we introduce the `query_counter` helper. This helper is made part of the name of the query.

```
helper def: query_counter : Integer = 1;
```

This rule also contains an imperative block, this time containing two variables. The first variable is used in the function call to the query, the other in the query itself.

```
    using
    {
        expressions1 : MMJess!Expression =
thisModule.Expression2Expression(ecadl_select.expression->at(2));
        expressions2 : MMJess!Expression = ecadl_select.expression-
>collect(e|thisModule.Expression2Expression(e));
    }
```

The target model element below uses the `query_counter` mentioned above to construct a unique name.

```
jess_expression_query_name1 : MMJess!Expression
(
    value <- 'query-' + thisModule.query_counter.toString()
),
```

By introducing a target model element that has no links to previous target model elements, we can create a new element under the root, i.e., without references to other elements. Such an element is necessary to create the Jess query outside of the Jess rule. Unfortunately, ATL makes no guarantee about the order of root elements. This means that the Jess query may only be created *after* the Jess rule that uses it. This problem has to be solved manually by placing the Jess query *before* the Jess rule that uses it.

```
jess_query : MMJess!Function
(
    name <- 'defquery',
    value <- '',
    expression <- Sequence
          {
              jess_expression_query_name2,
              jess_expression_declare,
              jess_expression_entities
          }
),
```

Finally, every time this transformation rule is called, the `query_counter` variable is incremented, so that the next transformation rule creates a new name.

```
do
{
    thisModule.query_counter <- thisModule.query_counter + 1;
}
```

5.2.3. Lifetime to Condition and Function

The output generated by the Lifetime function depends on the type of the Lifetime. Because of this, we have developed a different mapping for type "once" and for type <n>. For the mapping for lifetimes with type "once" we have developed the transformation rules `LifetimeOnce2LHS` and `LifetimeOnce2RHS`. For the mapping for lifetimes with type <n> we have developed the transformation rules `LifetimeN2LHS` and `LifetimeN2RHS`.

We do not elaborate on these transformation rules, because they do not use any constructs not already explained. However, we do present the way these transformation rules are called, namely through the use of helper functions. A helper function is a function, defined in the context of a specific metamodel class, which can, e.g., be used to group a large collection of elements, or filter out unwanted elements.

In the code below, the helper functions return value depends on the type of the lifetime. If this type is "once", then the transformation rule `LifetimeOnce2LHS` is called. If this type is <n>, then the transformation rule `LifetimeN2LHS` is called. If the ECA-DL rule does not contain a lifetime, then an empty Sequence is returned.

Finally, because we may return an (empty) Sequence in another Sequence, we need to use the `flatten()` to merge all the elements of the nested structure into a single Sequence.

```
helper context MMECADL!Rule def: extendLifetimeLHS(jess_condition :
MMJess!Condition) : MMJess!Condition =
    Sequence
      {
        -- Process Lifetime
        if not self.lifetime.oclIsUndefined() then
            if self.lifetime.lifetime = 'once' then
                    thisModule.LifetimeOnce2LHS(self.lifetime)
            else
                if self.lifetime.lifetime.toInteger() > 1 then
                  thisModule.LifetimeN2LHS(self.lifetime)
                else
                  Sequence{}
                endif
            endif
        else
            Sequence{}
        endif
      }->flatten();
```

The helper for the RHS is similar, and can be found in Appendix C.

5.2.4. CompoundTransition to Expression

The following mapping takes a CompoundTransition and transforms it to an Expression. Since a CompoundTransition can have both other CompoundTransitions as well as normal Transitions as its children, both have to be combined as children of the Expression to be created. The mapping is as follows:

If the reference `compoundtransition` is not undefined, i.e., when it is defined, then we can iterate over it. We recursively process CompoundTransitions.

```
lazy rule CompoundTransition2Expression
{
    ...
    expression <- Sequence
    {
        if not
ecadl_compoundtransition.compoundtransition.oclIsUndefined() then
            ecadl_compoundtransition.compoundtransition-
>collect(e|thisModule.CompoundTransition2Expression(e))
```

When we encounter Transitions, we first check their type, and depending on their type, call a specific function. E.g., `TransitionFalse2Function` places the NOT operator around the transition.

```
    ecadl_compoundtransition.transition->collect
    (e |
        if e.type = 'EnterTrue' or e.type = 'FalseToTrue' then
          thisModule.TransitionTrue2Expression(e)
        else
          if e.type = 'EnterFalse' or e.type = 'TrueToFalse' then
            thisModule.TransitionFalse2Function(e)
          else -- Changed
            thisModule.TransitionChanged2Function(e)
          endif
        endif
    )
```

5.2.5. Transition to Expression

As shown in Section 4.1.3, Transitions can be mapped to Expressions in different ways, depending on the type of the Transition. Below we show the mapping used to transform Transitions that have type “Changed” (i.e., they trigger when the event or the negation of the event happens).

Because in Jess the OR operator is a function, we also output a Function in our Jess model. This is possible, because Function inherits from Expression.

```
lazy rule TransitionChanged2Function
{
    from
        ecadl_transition : MMECADL!Transition
            (ecadl_transition.type = 'Changed')
    to
        jess_function : MMJess!Function
```

This mapping is one-to-many. A Transition of type ‘Changed’ is transformed to form a Function, which contains both the event and the negation of the event.

```
        (
            name <- 'or',
            value <- '',
            expression <- Sequence
                {
                    thisModule.Expression2Expression(
ecadl_transition.expression),
                    jess_function_neg
                }
        ),
        jess_function_neg : MMJess!Function
        (
            name <- 'not',
            value <- '',
            expression <-
thisModule.Expression2Expression(ecadl_transition.expression)
        )

}
```

6. Use case: Pete and Mary

In this chapter we present a use case that shows how the transformation process works.

In order to get this process working, we combine the transformation specifications from Chapter 5 and Appendix C, and import them in the ADT environment. In this way we have obtained a prototype transformation application, ready to transform input models.

We present how an ECA-DL rule can be transformed to a Jess rule by presenting a use case. This use case uses the same example used throughout this thesis, involving Pete and Mary:

When Mary enters the room, and Pete is already there, act surprised.

6.1. ECA-DL model

In ECA-DL, the example from this use case is written as follows:

```
Upon EnterTrue(inRoom(Mary))
When inRoom(Pete)
Do actSurprised()
```

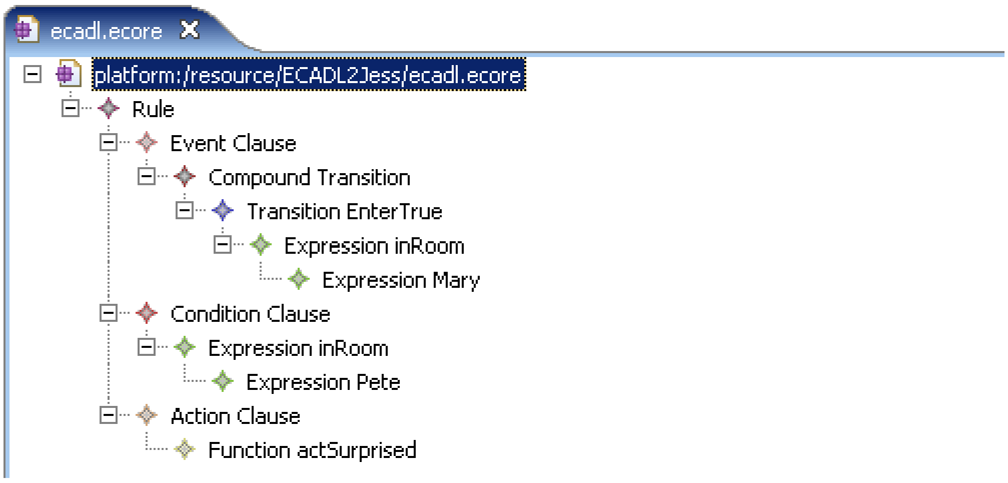
Transforming this ECA-DL rule to its corresponding model requires mapping the separate parts of the rule to elements in the ECA-DL metamodel. We thus map the `Upon` part to the `ConditionClause`, `EnterTrue` to `Transition`, `Mary` to `Expression`, etc. This step is done by hand.

In XMI, the language the tools use to represent models, the ECA-DL model looks as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Rule xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns="MMECADL">
  <eventclause>
    <compoundtransition>
      <transition type="EnterTrue">
        <expression value="inRoom">
          <expression value="Mary"/>
        </expression>
      </transition>
    </compoundtransition>
  </eventclause>
  <conditionclause>
      <expression value="inRoom">
        <expression value="Pete"/>
      </expression>
  </conditionclause>
  <actionclause>
    <function name="actSurprised" value="" />
  </actionclause>
</Rule>
```

This XMI code shows the different parts of the ECA-DL rule, as well as their relations to the metamodel. From the element “Rule” in the ECA-DL metamodel we follow the reference "eventclause", then the reference "compoundtransition", etc. Where applicable, the attributes of an element have been filled in.

The ATL Development Tools also support a graphical representation of models. Before we can use this, we must register the ECA-DL metamodel. We right-click the KM3 metamodel and inject it into an Ecore metamodel. We then right-click the generated Ecore metamodel, and Register it. We do the same for the Jess metamodel. Now, we can open the ECA-DL model in the Sample Ecore Model Editor.



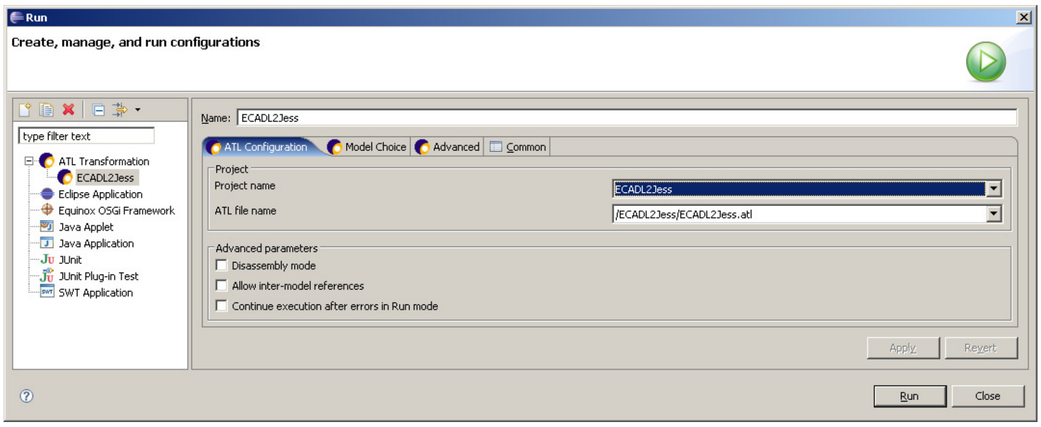
```
platform:/resource/ECADL2Jess/ecadl.ecore
  Rule
    Event Clause
      Compound Transition
        Transition EnterTrue
          Expression inRoom
            Expression Mary
    Condition Clause
      Expression inRoom
        Expression Pete
    Action Clause
      Function actSurprised
```

Figure 14: Graphical representation of the ECA-DL model

Before starting the transformation, we need to verify that this file is indeed a valid file, i.e., that it conforms to the ECA-DL metamodel. We right-click the Rule shown in Figure 14 and choose "Validate". The popup tells us that our model is valid and conforms to the ECA-DL metamodel.

6.2. Transforming the model

The next step is the actual transformation between the ECA-DL model and the Jess model. For this, we need to create a new transformation. From the menu, we choose "Run" => "Run…", and we create a new ATL Transformation, as shown in Figure 15.



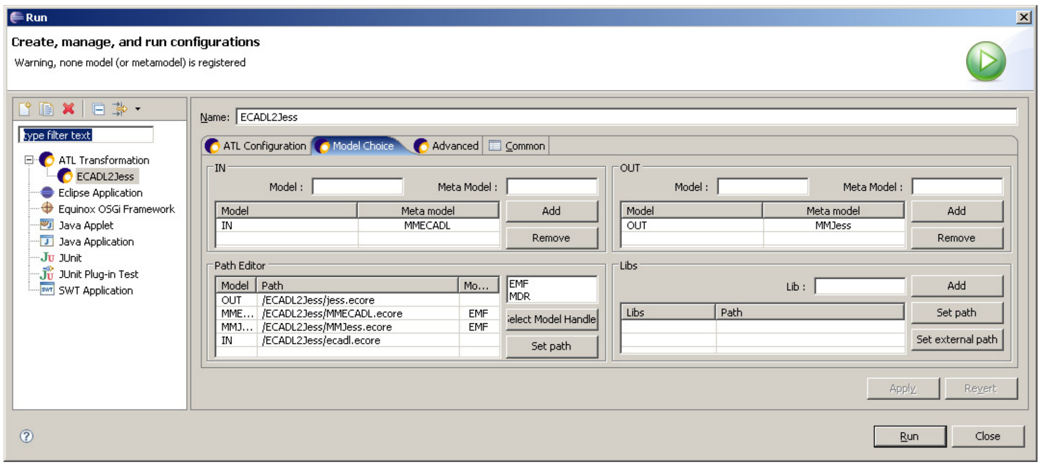
Name: ECADL2Jess

Project
Project name: ECADL2Jess
ATL file name: /ECADL2Jess/ECADL2Jess.atl

Advanced parameters
- [ ] Disassembly mode
- [ ] Allow inter-model references
- [ ] Continue execution after errors in Run mode

Figure 15: Configuring a new ATL Transformation

In the Model Choice tab, we fill in the models, and the metamodels they conform with, as shown in Figure 16.



Create, manage, and run configurations

Warning, none model (or metamodel) is registered

type filter text

Name: ECADL2Jess

IN

Model : Meta Model :

Model	Meta model
IN	MMECADL

OUT

Model : Meta Model :

Model	Meta model
OUT	MMJess

Path Editor

Model	Path	Mo...
OUT	/ECADL2Jess/jess.ecore	
MME...	/ECADL2Jess/MMECADL.ecore	EMF
MMJ...	/ECADL2Jess/MMJess.ecore	EMF
IN	/ECADL2Jess/ecadl.ecore	

EMF
MDR

Libs

Lib :

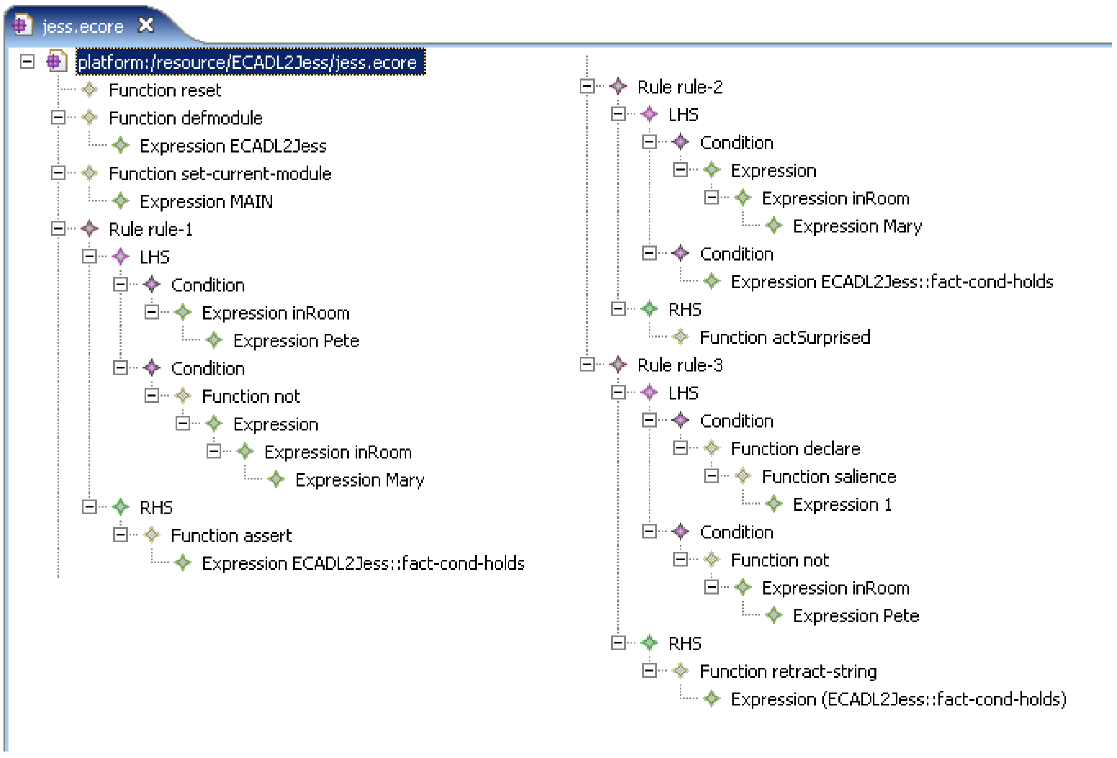
Libs	Path

Figure 16: Choosing models for the ATL transformation

We then click “Run” once more to execute the transformation.

6.3. Jess model

In the File navigator, we see that the file “jess.ecore” has been generated. Since we have already registered the Jess metamodel, we can open this file in the Sample Ecore Model Editor, as shown in Figure 17.



```
platform:/resource/ECADL2Jess/jess.ecore
  Function reset
  Function defmodule
    Expression ECADL2Jess
  Function set-current-module
    Expression MAIN
  Rule rule-1
    LHS
      Condition
        Expression inRoom
          Expression Pete
      Condition
        Function not
          Expression
            Expression inRoom
              Expression Mary
    RHS
      Function assert
        Expression ECADL2Jess::fact-cond-holds
  Rule rule-2
    LHS
      Condition
        Expression
          Expression inRoom
            Expression Mary
      Condition
        Expression ECADL2Jess::fact-cond-holds
    RHS
      Function actSurprised
  Rule rule-3
    LHS
      Condition
        Function declare
          Function salience
            Expression 1
      Condition
        Function not
          Expression inRoom
            Expression Pete
    RHS
      Function retract-string
        Expression (ECADL2Jess::fact-cond-holds)
```

Figure 17: Graphical representation of the Jess model

Figure 17 first shows the functions that reset the Jess engine and add a separate namespace. Below that, the three Jess rules are visible. These elements are also visible in the textual version of the Jess model, which can be found in Appendix B.

The last thing left to do is to map the Jess model to its textual representation, i.e., to a proper Jess rule. We simply serialize the information from the model above, placing parentheses where necessary, and obtain the Jess rule below. This step has been done by hand.

```
(reset)
(defmodule ECADL2Jess)
(set-current-module MAIN)
(defrule rule-1
  (inRoom Pete)
  (not (inRoom Mary))
  =>
  (assert (ECADL2Jess::fact-cond-holds))
)
(defrule rule-2
  (inRoom Mary)
  (ECADL2Jess::fact-cond-holds)
  =>
  (actSurprised)
)
(defrule rule-3
  (declare (salience 1))
  (not (inRoom Pete))
  =>
  (retract-string "(ECADL2Jess::fact-cond-holds)")
)
```

In Section 4.1.2, we have used the same example as above, and have shown what the Jess code should look like after transformation. The Jess code above is only slightly different from the code in Section 4.1.2. The only differences are the commands at the top of the Jess code and the namespace of the variables, which are both added later in Section 4.1.2.

This use case has shown that the example ECA-DL rule presented can indeed be properly automatically transformed to Jess.

7. Conclusions

In this chapter we draw conclusions from this thesis regarding the original goals and identify what future work can be done in the scope of this project.

7.1. Mapping from ECA-DL to Jess

Because of the different nature of ECA-DL and the Jess language, and the different possibilities these languages offer, it proved quite a challenge to develop the mapping between ECA-DL and Jess.

Although the structures of ECA-DL and Jess rules are identical in simple cases, like the example in Section 4.1.1, more complex ECA-DL rules require a more complex mapping.

The main problem has been to properly represent the difference between an ECA-DL event and a condition in Jess. This problem has been solved by splitting up the ECA-DL rule into three Jess rules.

The only problem in the mapping that has not been solved involves the use of the Scope construct in combination with a Lifetime. In ECA-DL, a Lifetime works on a per-rule basis. When a Scope construct is placed around this rule, then the ECA-DL semantics state that every rule still has its own Lifetime. However, in our mapping of the Scope construct to Jess, the Scope construct merges with the rule, so that the Lifetime now works over this single created rule, instead of over the original set of rules.

Unfortunately, the mapping is not complete. We have not been able to find correct mappings for the following ECA-DL constructs:

- Support for the value “unknown”.
- The lifetimes `from <start> to <end>`, `to <end>` and `frequency <n> times per <period>`.

These constructs are thus not supported by the transformation.

7.2. ATL transformation environment

Once the Eclipse environment has been set up and all the files are in place, the transformation process runs smoothly. The environment itself, however, is rather static.

The files to be transformed and the metamodels to be used are specified once in the run dialogue of the ATL transformation dialogue. If we want to transform another file, we need to change the ATL transformation, or create an entire new one.

Also, when one of the files contains an error, the error messages can be rather cryptic. For example, when the input model does not conform to the metamodel, and we try to launch the transformation, the error message shows “An internal error occurred during: "Launching".” instead of pointing out that the input model is incorrect.

However, there may be an even bigger drawback about the ATL Development Tools.

If, on the one hand, ECA-DL rules have to be used in the deployment environment, then they also need to be transformed to Jess rules during deployment. However, it is only possible to run the transformation process inside the Eclipse environment. This seriously inhibits the use of this transformation process in context-aware applications. Such applications typically run on small computing devices, which are unable to run such an extensive development platform.

If, on the other hand, ECA-DL rules are already transformed to Jess rules during development, then this is no longer a problem.

7.3. ATL transformation process

The transformation process in Eclipse works quite well. We have identified only two problems.

1. In the extended ECA-DL metamodel from Section 3.1.3, ActionClause links to Function by means of an association. In an ECA-DL model, it is thus possible for an ActionClause to point to a Function. However, because when transforming an ECA-DL rule, the ADT only follows composition links, it does not see this Function, and thus does not transform it either. We have solved this problem by converting all association links to composition links. This is possible, because in a model, the ActionClause indeed contains a Function.

2. When an ECA-DL rule is being transformed to a Jess rule, first the Jess rule itself is constructed, then the LHS and RHS, containing conditions and functions, and so forth.

   If a Select construct is found in the Action part of an ECA-DL rule, then this construct is mapped to the RHS of the Jess rule and to a separate query function. The mapping in Chapter 4 states that this query function should be defined *before* the rule. However, since the Jess rule has already been created, the query function is placed *after* the rule. We have not been able to find a solution for this problem. When this problem occurs, the query function should be manually moved before the rule.

7.4. Automated transformations

It has been our goal in this project to create an automated transformation between ECA-DL rules and Jess rules. We have achieved this goal to a large extent. In the final intended transformation process, all 6 steps from Figure 2 and Figure 3 in Section 2.3 are performed to transform an ECA-DL rule to a corresponding Jess rule.

In this project, we have worked on steps 1, 2, 3 and 5. Step 3, developing the mapping between ECA-DL and Jess, is the only step that has not been completed, because the mapping is not complete and the transformation not completely automated.

We have not worked on steps 4 and 6, where an ECA-DL rule is transformed to its model counterpart, and the generated Jess model back to its textual counterpart.

As of this writing, the Eclipse plugin to transform between model and text has not been released yet. The alternative, writing our own transformation programs to transform between model and text would require too much work, and is thus beyond the scope of this project.

Therefore we had to manually transform ECA-DL rules to models that conform to the ECA-DL metamodel, and to manually transform Jess models back to their textual representation.

7.5. Future work

In this project, we have shown how ECA-DL rules can be transformed to Jess rules, albeit as models.

In Section 1.2, we stated our overall goal, i.e., to facilitate the implementation of context-aware applications that implement the ECA pattern, by providing a means for using ECA-DL rules in an already existing interpreter.

The following topics can bring us closer to this goal:

- The two main steps missing in the transformation process are, on the one hand, the transformation between ECA-DL rules as text and those as models, and on the other hand between Jess rules as models and those as text. By introducing model-to-text transformation, these two steps can become part of the overall transformation process. When the model-to-text transformation plugin mentioned in Section 7.4 is released, we advise to evaluate the usefulness of this plugin in the scope of this project and to evaluate if it is more beneficial to use this plugin or write dedicated parsers (for text to model) and serializers (for model to text);

- In order to use this complete transformation process in a context-aware application, we need to set up the transformation process as a module. This module can then be inserted into a context-aware application. This removes the need for the Eclipse environment;

- The current transformation only accepts ECA-DL rules as input, and always gives Jess rules as output. In order to make this transformation more generic, we can introduce a general rule language. The transformation process then consists of two steps:

    1. Source rules, e.g., written in ECA-DL, are transformed to generic rules. These generic rules do not need to be textual, but can be models only.

    2. These generic rules are transformed to target rules, e.g., rules written in Jess.

  It is then much easier to make additional mappings from or to other rule languages.

References

[1] Dockhorn Costa, P. et al., *Controlling Services in a Mobile Context-Aware Infrastructure*. Proc. Of the Second Workshop on Context Awareness for Proactive Systems (CAPS 2006), Kassel, Germany, June 2006

[2] Dockhorn Costa, P., Ferreira Pires, L., van Sinderen, M., *Architectural Patterns for Context-Aware Services Platforms*. Proc. of the Second International Workshop on Ubiquitous Computing (IWUC 2005), Miami, May 2005

[3] Etter, R., Dockhorn Costa, P., Broens, T., *A Rule-Based Approach Towards Context-Aware User Notification Services*. Proc. of the IEEE International Conference on Pervasive Services 2006, Lyon, France, June 2006

[4] Daniele, L.M. (2006), *Towards a Rule-based Approach for Context-Aware Applications*. MSc Thesis, University of Twente

[5] Object Management Group, *Model Driven Architecture*.
Available at: http://www.omg.org/mda/

[6] Dey, A.K. (2000), *Providing Architectural Support for Building Context-Aware Applications*. PhD Thesis, Georgia Institute of Technology

[7] Cebulla, M., *Transformation of Domain-specific Models as Foundation for Context-Awareness in Complex Systems*. Proceedings of the 5th OOPSLA Workshop on Domain-Specific Modeling (DSM'05), Tolvanen, J.-P., Sprinkle, J., Rossi, M., (eds.), Computer Science and Information System Reports, Technical Reports, TR-36, University of Jyväskylä, Finland 2005, ISBN 951-39-2202-2

[8] Kurtev, I. (2005), *Adaptability of Model Transformations*. PhD Thesis, University of Twente

[9] Iacob, M.E. (ed), Jonkers, H., van Dieten, W., *Model Transformations Case Studies*. Freeband project (A-MUSE/D2.6), 2005

[10] Almeida, J.P.A. et al., *Model-Driven Development of Context-Aware Services*. Distributed Applications and Interoperable Systems (DAIS 2006), 6th IFIP International Conference, Lecture Notes in Computer Science, vol. 4025, Springer, 2006

[11] Freeband AWARENESS project
http://awareness.freeband.nl

[12] Sandia National Laboratories, *Jess Rule Engine*,
Available at: http://herzberg.ca.sandia.gov/jess/

[13] Object Management Group, *XML Metadata Interchange*.
http://en.wikipedia.org/wiki/XML_Metadata_Interchange

[14] Object Management Group, *MetaObject Facility*.
Available at: http://www.omg.org/mof/

[15] Object Management Group, *Unified Modeling Language*.
Available at: http://www.uml.org/

[16] Rapela, D. (Ed.) (2004), *MDA modeling and application principles*. MODA-TEL Consortium, Deliverable 3.3
Available at: http://www.modatel.org/public/deliverables/D3.3.htm

[17] The Eclipse Foundation, *Eclipse Modeling Framework*.
Available at: http://www.eclipse.org/emf/

[18] The Eclipse Foundation, *ATLAS Transformation Language*.
Available at: http://www.eclipse.org/m2m/atl/

[19] ATLAS group, LINA & INRIA, *Kernel Meta Meta Model*. Version 0.3, August 2005.
Available at:
http://www.eclipse.org/gmt/am3/km3/doc/KernelMetaMetaModel%5Bv00.06%5D.pdf

[20] StarUML, *The Open Source UML/MDA Platform*.
Available at: http://www.staruml.com/

[21] ATLAS group, LINA & INRIA, *ATL Starter's Guide*. Version 0.1, December 2005.
Available at: http://www.eclipse.org/m2m/atl/doc/ATL_Starter_Guide.pdf

[22] ATLAS group, LINA & INRIA, *ATL User Manual*. Version 0.7, February 2006.
Available at: http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf

Appendix A – KM3 metamodels

In the metamodels below, all association links have been changed into composition links (`container`), because otherwise the ADT plugin refused to transform them.

ECA-DL metamodel

```
package MMECADL {

      class Rule {
            reference eventclause container : EventClause oppositeOf
rule ;
            reference conditionclause[0-1] container :
ConditionClause oppositeOf rule ;
            reference actionclause container : ActionClause
oppositeOf rule ;
            reference lifetime[0-1] container : Lifetime ;
            reference scope[0-1] : Scope oppositeOf rule ;
      }

      class EventClause {
            reference rule : Rule oppositeOf eventclause ;
            reference compoundtransition container :
CompoundTransition ;
      }

      class CompoundTransition {
            reference compoundtransition[0-2] container :
CompoundTransition ;
            reference transition[0-2] container : Transition ;
      }

      class Transition {
            reference expression container : Expression ;
            attribute type : String ;
      }

      class Expression {
            reference expression[*] container : Expression ;
            reference function[*] container : Function ;
            reference select[*] container : Select ;
            attribute value : String ;
      }

      class ConditionClause {
            reference rule : Rule oppositeOf conditionclause ;
            reference expression container : Expression ;
      }

      class ActionClause {
            reference rule : Rule oppositeOf actionclause ;
            reference function[1-*] container : Function ;
      }

      class Function extends Expression {
            attribute name : String ;
      }
      class Lifetime {
            attribute lifetime : String ;
      }

      class Scope {
            attribute var : String ;
            reference rule container : Rule oppositeOf scope ;
            reference select container : Select ;
      }

      class Select {
            attribute var : String ;
            reference expression[2-2] container : Expression ;
      }

}

package PrimitiveTypes {
      datatype String;
}
```

Jess metamodel

```
package MMJess {

      class Condition {
            reference expression container : Expression ;
      }

      class Expression {
            attribute value : String ;
            reference expression[*] container : Expression ;
            reference function[*] container : Function ;
      }

      class Function extends Expression {
            attribute name : String ;
            reference rhs[*] : RHS oppositeOf function ;
      }

      class LHS {
            reference condition[1-*] container : Condition ;
      }

      class RHS {
            reference function[1-*] container : Function oppositeOf
rhs ;
      }

      class Rule {
            attribute name : String ;
            reference lhs[0-1] container : LHS ;
            reference rhs[0-1] container : RHS ;
      }
}

package PrimitiveTypes {
  datatype String;
}
```

Appendix B – Ecore models

ECA-DL model

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Rule xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI" xmlns="MMECADL">
  <eventclause>
    <compoundtransition>
      <transition type="EnterTrue">
        <expression value="inRoom">
          <expression value="Mary"/>
        </expression>
      </transition>
    </compoundtransition>
  </eventclause>
  <conditionclause>
      <expression value="inRoom">
        <expression value="Pete"/>
      </expression>
  </conditionclause>
  <actionclause>
    <function name="actSurprised" value="" />
  </actionclause>
</Rule>
```

Jess model

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xmi:XMI xmi:version="2.0"
    xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="MMJess">
  <Function value="" name="reset"/>
  <Function value="" name="defmodule">
    <expression value="ECADL2Jess"/>
  </Function>
  <Function value="" name="set-current-module">
    <expression value="MAIN"/>
  </Function>
  <Rule name="rule-1">
    <lhs>
      <condition>
        <expression value="inRoom">
          <expression value="Pete"/>
        </expression>
      </condition>
      <condition>
        <expression xsi:type="Function" value="" name="not">
          <expression value="">
            <expression value="inRoom">
              <expression value="Mary"/>
            </expression>
          </expression>
        </expression>
      </condition>
    </lhs>
    <rhs>
      <function value="" name="assert">
        <expression value="ECADL2Jess::fact-cond-holds"/>
      </function>
    </rhs>
  </Rule>
  <Rule name="rule-2">
    <lhs>
      <condition>
        <expression value="">
          <expression value="inRoom">
            <expression value="Mary"/>
          </expression>
        </expression>
      </condition>
      <condition>
        <expression value="ECADL2Jess::fact-cond-holds"/>
      </condition>
    </lhs>
    <rhs>
      <function value="" name="actSurprised"/>
    </rhs>
  </Rule>
  <Rule name="rule-3">
    <lhs>
      <condition>
        <expression xsi:type="Function" value="" name="declare">
          <expression xsi:type="Function" value="" name="salience">
            <expression value="1"/>
          </expression>
        </expression>
      </condition>
      <condition>
        <expression xsi:type="Function" value="" name="not">
          <expression value="inRoom">
            <expression value="Pete"/>
          </expression>
        </expression>
      </condition>
    </lhs>
    <rhs>
      <function value="" name="retract-string">
        <expression value="(ECADL2Jess::fact-cond-holds)"/>
      </function>
    </rhs>
  </Rule>
</xmi:XMI>
```

Appendix C – ATL Transformation Specification

```
module ECADL2Jess; -- Module Template
create OUT : MMJess from IN : MMECADL;

------------------------------------------------------------
-- HELPERS                                                --
------------------------------------------------------------

-- Checks if a rule has a ConditionClause.
-- This attribute is evaluated only once!
helper context MMECADL!Rule def: hasCondition : Boolean =
    not self.conditionclause.oclIsUndefined();

-- Keeps track of the number of Select "queries".
helper def: query_counter : Integer = 1;

-- Introduce the Lifetime in the LHS.
helper context MMECADL!Rule def: extendLifetimeLHS(jess_condition :
MMJess!Condition) : MMJess!Condition =
    Sequence
    {
        -- Process Lifetime
        if not self.lifetime.oclIsUndefined() then
            if self.lifetime.lifetime = 'once' then
                thisModule.LifetimeOnce2LHS(self.lifetime)
            else
                if self.lifetime.lifetime.toInteger() > 1 then
                    thisModule.LifetimeN2LHS(self.lifetime)
                else
                    Sequence{}
                endif
            endif
        else
            Sequence{}
        endif
    }->flatten();

-- Introduce the Lifetime in the RHS.
helper context MMECADL!Rule def: extendLifetimeRHS(jess_function :
MMJess!Function) : MMJess!Function =
    Sequence
    {
        -- Process Lifetime
        if not self.lifetime.oclIsUndefined() then
            if self.lifetime.lifetime = 'once' then
                thisModule.LifetimeOnce2RHS(self.lifetime)
            else
                if self.lifetime.lifetime.toInteger() > 1 then
                    thisModule.LifetimeN2RHS(self.lifetime)
                else
                    Sequence{}
                endif
            endif
        else
            Sequence{}
        endif
    }>flatten();

-------------------------------------------------------------
-- BASIC MAPPING                                           --
-------------------------------------------------------------

-------------------------------------------------------------
-- Convert main rule with condition.                       --
--                                                         --
-- This rule is applied when the rule                      --
-- does contain a Condition part.                          --
-------------------------------------------------------------
rule Rule2RuleComplex
{
    from
        ecadl_rule : MMECADL!Rule
          (ecadl_rule.hasCondition)
    to
        -- Always issue reset. This is necessary for
        -- Jess to understand the absence of Facts.
        jess_reset : MMJess!Function
        (
            name <- 'reset',
            value <- ''
        ),

        -- Create separate namespace for variables.
        jess_namespace : MMJess!Function
        (
            name <- 'defmodule',
            value <- '',
            expression <- jess_namespace_ECADL2Jess
        ),
        jess_namespace_ECADL2Jess : MMJess!Expression
        (
            value <- 'ECADL2Jess'
        ),

        -- Switch back to default module.
        jess_module : MMJess!Function
        (
            name <- 'set-current-module',
            value <- '',
            expression <- jess_module_MAIN
        ),
        jess_module_MAIN : MMJess!Expression
        (
            value <- 'MAIN'
        ),


        -------------------------------------------------------------
        -- Create the 3 rules.
        -------------------------------------------------------------
        jess_rule1 : MMJess!Rule
        (
            name <- 'rule-1',
            lhs <- jess_lhs1,
            rhs <- jess_assert_fact
        ),
        jess_rule2 : MMJess!Rule
        (
            name <- 'rule-2',
            lhs <- jess_lhs2,
            rhs <- jess_action
        ),
        jess_rule3 : MMJess!Rule
        (
            name <- 'rule-3',
            lhs <- jess_lhs3,
            rhs <- jess_retract_fact
        ),


        ------------------------------------------------------------
        -- Create the LHS's
        ------------------------------------------------------------
        -- Create first LHS, containing condition and negation of
event.
        jess_lhs1 : MMJess!LHS
        (
            condition <- Sequence
                {
                    -- Transform Scope.
                    if not ecadl_rule.scope.oclIsUndefined() then
                        thisModule.Scope2Condition(ecadl_rule.scope)
                    else
                        Sequence{}
                    endif,
                    jess_condition,
                    jess_neg_event
                }
        ),
        -- Create second LHS, containing event and Fact.
        jess_lhs2 : MMJess!LHS
        (
            condition <- Sequence
                {
                    -- Transform Scope.
                    if not ecadl_rule.scope.oclIsUndefined() then
                        thisModule.Scope2Condition(ecadl_rule.scope)
                    else
                        Sequence{}
                    endif,
                    jess_event,
                    jess_check_fact
                }
        ),
        -- Create third LHS, containing negation of condition.
        jess_lhs3 : MMJess!LHS
        (
            condition <- Sequence
                {
                    jess_salience,
                    -- Transform Scope.
                    if not ecadl_rule.scope.oclIsUndefined() then
                        thisModule.Scope2Condition(ecadl_rule.scope)
                    else
                        Sequence{}
                    endif,
                    jess_neg_condition
                }
        ),


        ------------------------------------------------------------
        -- Create the conditions (fill in the LHS's)
        ------------------------------------------------------------
        -- Create event.
        jess_event : MMJess!Condition
        (
            expression <-
thisModule.CompoundTransition2Expression(ecadl_rule.eventclause.compo
undtransition)
        ),
        -- Create negation of event.
        jess_neg_event : MMJess!Condition
        (
            expression <- jess_neg_event_function
        ),
        jess_neg_event_function : MMJess!Function
        (
            name <- 'not',
            value <- '',
            expression <-
thisModule.CompoundTransition2Expression(ecadl_rule.eventclause.compo
undtransition)
        ),
        -- Create condition.
        jess_condition : MMJess!Condition
        (
            expression <-
thisModule.Expression2Expression(ecadl_rule.conditionclause.expressio
n)
        ),
        -- Create negation of condition.
        jess_neg_condition : MMJess!Condition
        (
            expression <- jess_neg_condition_function
        ),
        jess_neg_condition_function : MMJess!Function
        (
            name <- 'not',
            value <- '',
            expression <-
thisModule.Expression2Expression(ecadl_rule.conditionclause.expressio
n)
        ),


        ------------------------------------------------------------
        -- Create declaration of salience.
        ------------------------------------------------------------
        jess_salience : MMJess!Condition
        (
            expression <- jess_salience_declare
        ),
        jess_salience_declare : MMJess!Function
        (
            name <- 'declare',
            value <- '',
            expression <- jess_salience_salience
        ),
        jess_salience_salience : MMJess!Function
        (
            name <- 'salience',
            value <- '',
            expression <- jess_salience_number
        ),
        jess_salience_number : MMJess!Expression
        (
            value <- '1'
        ),


        ------------------------------------------------------------
        -- Create action.
        ------------------------------------------------------------
        jess_action : MMJess!RHS
        (
            function <- ecadl_rule.actionclause.function-
>collect(e|thisModule.Function2Function(e))
        ),


        ------------------------------------------------------------
        -- Create conditions/actions for Fact.
        ------------------------------------------------------------
        jess_assert_fact : MMJess!RHS
        (
            function <- jess_assert_fact_function
        ),
        jess_assert_fact_function : MMJess!Function
        (
            name <- 'assert',
            value <- '',
            expression <- jess_assert_fact_expression
        ),
        jess_assert_fact_expression : MMJess!Expression
        (
            value <- 'ECADL2Jess::fact-cond-holds'
        ),
        jess_check_fact : MMJess!Condition
        (
            expression <- jess_check_fact_expression
        ),
        jess_check_fact_expression : MMJess!Expression
        (
            value <- 'ECADL2Jess::fact-cond-holds'
        ),
        jess_retract_fact : MMJess!RHS
        (
            function <- jess_retract_fact_function
        ),
        jess_retract_fact_function : MMJess!Function
        (
            name <- 'retract-string',
            value <- '',
            expression <- jess_retract_fact_expression
        ),
        jess_retract_fact_expression : MMJess!Expression
        (
            value <- '(ECADL2Jess::fact-cond-holds)'
        )
```

```
    do
    {
        -- Introduce Lifetime.
        jess_lhs2.condition <-
ecadl_rule.extendLifetimeLHS(jess_lhs2.condition);
        jess_action.function <-
ecadl_rule.extendLifetimeRHS(jess_action.function);
    }
}

--------------------------------------------------------------
-- Convert main rule without condition.                     --
--                                                          --
-- This rule is applied when the rule                       --
-- does not contain a Condition part.                       --
--------------------------------------------------------------
rule Rule2RuleSimple
{
    from
        ecadl_rule : MMECADL!Rule
            (not ecadl_rule.hasCondition)
    to
        -- Always issue reset. This is necessary for
        -- Jess to understand the absence of Facts.
        jess_function : MMJess!Function
        (
            name <- 'reset',
            value <- ''
        ),
        -- Create main rule.
        jess_rule : MMJess!Rule
        (
            name <- 'rule-1',
            lhs <- jess_lhs,
            rhs <- jess_rhs
        ),

        -- Create LHS and its condition.
        jess_lhs : MMJess!LHS
        (
            condition <- Sequence
                {
                    -- Transform Scope.
                    if not ecadl_rule.scope.oclIsUndefined() then
                        thisModule.Scope2Condition(ecadl_rule.scope)
                    else
                        Sequence{}
                    endif,
                    -- Transform Conditions.
                    jess_condition
                }
        ),
        jess_condition : MMJess!Condition
        (
            expression <-
thisModule.CompoundTransition2Expression(ecadl_rule.eventclause.compo
undtransition)
        ),

        -- Create RHS.
        jess_rhs : MMJess!RHS
        (
            function <- ecadl_rule.actionclause.function-
>collect(e|thisModule.Function2Function(e))
        )
    do
    {
        -- Introduce Lifetime.
        jess_lhs.condition <-
ecadl_rule.extendLifetimeLHS(jess_lhs.condition);
        jess_rhs.function <-
ecadl_rule.extendLifetimeRHS(jess_rhs.function);
    }
}


------------------------------------------------------------
-- END OF BASIC MAPPING                                   --
------------------------------------------------------------

------------------------------------------------------------
-- DETAILED MAPPING                                       --
------------------------------------------------------------

-- This rule copies all the children from a CompoundTransition to an
Expression.
-- CompoundTransitions contain no data by themselves.
lazy rule CompoundTransition2Expression
{
    from
        ecadl_compoundtransition : MMECADL!CompoundTransition
    to
        -- Transform both CompoundTransitions and Transitions to
Expressions.
        jess_expression : MMJess!Expression
        (
            value <- '',
            expression <- Sequence
                {
                    if not
ecadl_compoundtransition.compoundtransition.oclIsUndefined() then
                        ecadl_compoundtransition.compoundtransition-
>collect(e|thisModule.CompoundTransition2Expression(e))
                    else
                        Sequence{}
                    endif,
                    if not
ecadl_compoundtransition.transition.oclIsUndefined() then
                        ecadl_compoundtransition.transition->collect
                            (e |
                                if e.type = 'EnterTrue' or e.type =
'FalseToTrue' then

thisModule.TransitionTrue2Expression(e)
                                else
                                    if e.type = 'EnterFalse' or
e.type = 'TrueToFalse' then

thisModule.TransitionFalse2Function(e)
                                    else -- Changed

thisModule.TransitionChanged2Function(e)
```

```
                                                endif
                                            endif
                                        )
                                else
                                    Sequence{}
                                endif
                            }->flatten()
                )
}

-- These rules then transform the Transitions themselves.
-- Transform EnterTrue()/FalseToTrue().
lazy rule TransitionTrue2Expression
{
    from
        ecadl_transition : MMECADL!Transition
            (ecadl_transition.type = 'EnterTrue' or
ecadl_transition.type = 'FalseToTrue')
    to
        jess_expression : MMJess!Expression
        (
            -- Do not make the Transition itself into an Expression.
            -- Just copy the Expression child of Transition.
            -- This is possible due to the one-to-one relationship,
            -- meaning a Transition _always_ has an Expression.
            -- In this way we model leaving out the EnterTrue
            -- transition completely.
            value <- ecadl_transition.expression.value,
            expression <- if not
ecadl_transition.expression.expression.oclIsUndefined() then
                            ecadl_transition.expression.expression-
>collect(e|thisModule.Expression2Expression(e))
                          else
                            Sequence{}
                          endif,
            -- Copy functions.
            function <- if not
ecadl_transition.expression.function.oclIsUndefined() then
                          ecadl_transition.expression.function-
>collect(e|thisModule.Function2Function(e))
                        else
                          Sequence{}
                        endif
        )
}
-- Transform EnterFalse()/TrueToFalse().
lazy rule TransitionFalse2Function
{
    from
        ecadl_transition : MMECADL!Transition
            (ecadl_transition.type = 'EnterFalse' or
ecadl_transition.type = 'TrueToFalse')
    to
        jess_function : MMJess!Function
        (
            name <- 'not',
            value <- '',
            expression <-
thisModule.Expression2Expression(ecadl_transition.expression)
        )
}
```

```
-- Transform Changed().
lazy rule TransitionChanged2Function
{
    from
        ecadl_transition : MMECADL!Transition
            (ecadl_transition.type = 'Changed')
    to
        jess_function : MMJess!Function
        (
            name <- 'or',
            value <- '',
            expression <- Sequence
                {

thisModule.Expression2Expression(ecadl_transition.expression),
                    jess_function_neg
                }
        ),
        jess_function_neg : MMJess!Function
        (
            name <- 'not',
            value <- '',
            expression <-
thisModule.Expression2Expression(ecadl_transition.expression)
        )
}

lazy rule Expression2Expression
{
    from
        ecadl_expression : MMECADL!Expression
            -- Do not match Functions, which inherit from Expression.
            (ecadl_expression.oclIsTypeOf(MMECADL!Expression))
    to
        jess_expression : MMJess!Expression
        (
            value <- ecadl_expression.value,
            -- Copy expressions.
            expression <- if not
ecadl_expression.expression.oclIsUndefined() then
                                ecadl_expression.expression-
>collect(e|thisModule.Expression2Expression(e))
                              else
                                Sequence{}
                              endif,
            function <- Sequence
                {
                    -- Copy expressions.
                    if not ecadl_expression.function.oclIsUndefined()
then
                        ecadl_expression.function-
>collect(e|thisModule.Function2Function(e))
                    else
                        Sequence{}
                    endif,
                    -- Copy selects.
                    if not ecadl_expression.select.oclIsUndefined()
then
                        ecadl_expression.select-
>collect(e|thisModule.Select2Function(e))
                    else
```

```
                        Sequence{}
                    endif
                }->flatten()
        )
}

lazy rule Function2Function
{
    from
        ecadl_function : MMECADL!Function
    to
        jess_function : MMJess!Function
        (
            name <- ecadl_function.name,
            value <- ecadl_function.value,
            -- Copy expressions.
            expression <- if not
ecadl_function.expression.oclIsUndefined() then
                            ecadl_function.expression-
>collect(e|thisModule.Expression2Expression(e))
                          else
                            Sequence{}
                          endif,
            function <- Sequence
                {
                    -- Copy expressions.
                    if not ecadl_function.function.oclIsUndefined()
then
                        ecadl_function.function-
>collect(e|thisModule.Function2Function(e))
                    else
                        Sequence{}
                    endif,
                    -- Copy selects.
                    if not ecadl_function.select.oclIsUndefined()
then
                        ecadl_function.select-
>collect(e|thisModule.Select2Function(e))
                    else
                        Sequence{}
                    endif
                }->flatten()
        )
}

------------------------------------------------------------
-- END OF DETAILED MAPPING                                --
------------------------------------------------------------

------------------------------------------------------------
-- EXTENDED MAPPING                                       --
------------------------------------------------------------

-- Transform Scope and Select.
lazy rule Scope2Condition
{
    from
        ecadl_scope : MMECADL!Scope
    using
    {
        -- Without this variable, the Expressions are transformed
again for some reason...
        expressions : MMJess!Expression =
ecadl_scope.select.expression-
>collect(e|thisModule.Expression2Expression(e));
    }
    to
        jess_condition : MMJess!Condition
        (
            expression <- jess_expression
        ),
        jess_expression : MMJess!Expression
        (
            value <- '',
            expression <- Sequence
                {
                    -- First element of ecadl_scope.expression[2]
                    expressions->at(1),
                    jess_expression_var,
                    -- Second element of ecadl_scope.expression[2]
                    expressions->at(2)
                }
        ),
        jess_expression_var : MMJess!Expression
        (
            value <- ecadl_scope.var
        )
}

-- Transform Select.
lazy rule Select2Function
{
    from
        ecadl_select : MMECADL!Select
    using
    {
        -- Without these variables, the Expressions are transformed
again for some reason...
        -- And yes, we need these twice, because we use them twice.
        expressions1 :
thisModule.Expression2Expression(ecadl_select.expression->at(2));
        expressions2 : MMJess!Expression = ecadl_select.expression-
>collect(e|thisModule.Expression2Expression(e));
    }
    to
        jess_function : MMJess!Function
        (
            name <- 'run-query*',
            value <- '',
            expression <- Sequence
                {
                    jess_expression_query_name1,
                    expressions1->at(2)
                }
        ),
        jess_expression_query_name1 : MMJess!Expression
        (
            value <- 'query-' + thisModule.query_counter.toString()
        ),

        -- Also create separate command to initialize defquery.
        jess_query : MMJess!Function
        (
            name <- 'defquery',
            value <- '',
            expression <- Sequence
                {
                    jess_expression_query_name2,
                    jess_expression_declare,
                    jess_expression_entities
                }
        ),
        -- Query name.
        jess_expression_query_name2 : MMJess!Expression
        (
            value <- 'query-' + thisModule.query_counter.toString()
        ),
        -- Declare.
        jess_expression_declare : MMJess!Expression
        (
            value <- '',
            expression <- Sequence
                {
                    jess_expression_declare_declare,
                    jess_expression_declare_variables,
                    jess_expression_declare_selectvar
                }
        ),
        jess_expression_declare_declare : MMJess!Expression
        (
            value <- 'declare'
        ),
        jess_expression_declare_variables : MMJess!Expression
        (
            value <- 'variables'
        ),
        jess_expression_declare_selectvar : MMJess!Expression
        (
            value <- ecadl_select.var
        ),
        -- Entities.
        jess_expression_entities : MMJess!Expression
        (
            value <- '',
            expression <- Sequence
                {
                    -- First element of ecadl_scope.expression[2]
                    expressions2->at(1),
                    jess_expression_entities_selectvar,
                    expressions2->at(2)
                }
        ),
        jess_expression_entities_selectvar : MMJess!Expression
        (
            value <- ecadl_select.var
        )
    do
    {
        thisModule.query_counter <- thisModule.query_counter + 1;
    }
}
```

```
-- Transform Lifetime.
-- Transform Lifetime 'once'.
lazy rule LifetimeOnce2LHS
{
    from
        ecadl_lifetime : MMECADL!Lifetime
    to
        jess_condition : MMJess!Condition
        (
            expression <- jess_function
        ),
        jess_function : MMJess!Function
        (
            name <- 'not',
            value <- '',
            expression <- jess_lexpression
        ),
        jess_lexpression : MMJess!Expression
        (
            value <- '(ECADL2Jess::rule-1-fired)'
        )
}
lazy rule LifetimeOnce2RHS
{
    from
        ecadl_lifetime : MMECADL!Lifetime
    to
        jess_function : MMJess!Function
        (
            name <- 'assert',
            value <- '',
            expression <- jess_expression
        ),
        jess_expression : MMJess!Expression
        (
            value <- '(ECADL2Jess::rule-1-fired)'
        )
}
-- Transform Lifetime n.
lazy rule LifetimeN2LHS
{
    from
        ecadl_lifetime : MMECADL!Lifetime
    to
        jess_condition : MMJess!Condition
        (
            expression <- jess_function
        ),
        jess_function : MMJess!Function
        (
            name <- 'test',
            value <- '',
            expression <- jess_function_comp
        ),
        jess_function_comp : MMJess!Function
        (
            name <- '>',
            value <- '',
            expression <- Sequence{jess_expression_var,
jess_expression_null}
        ),
        jess_expression_var : MMJess!Expression
        (
            value <- '?*rule-1-count*'
        ),
        jess_expression_null : MMJess!Expression
        (
            value <- '0'
        ),

        -- Also create separate command to initialize variable.
           jess_init_function : MMJess!Function
           (
               name <- 'bind',
               value <- '',
               expression <- Sequence{jess_expression_init_var,
jess_expression_int}
           ),
           jess_expression_init_var : MMJess!Expression
           (
               value <- '?*rule-1-count*'
           ),
           jess_expression_int : MMJess!Expression
           (
               value <- ecadl_lifetime.lifetime
           )
}
lazy rule LifetimeN2RHS
{
    from
        ecadl_lifetime : MMECADL!Lifetime
    to
        jess_function : MMJess!Function
        (
            name <- '--',
            value <- '',
            expression <- jess_expression
        ),
        jess_expression : MMJess!Expression
        (
            value <- '?*rule-1-count*'
        )
}

------------------------------------------------------------
-- END OF EXTENDED MAPPING                                 --
------------------------------------------------------------
```